USENIX

SAN DIEGO CONFERENCE PROCEEDINGS

WINTER

1993

WINTER 1993
TECHNICAL
CONFERENCE

USENIX

SAN DIEGO
CALIFORNIA
JANUARY 25–29

CONFERENCE PROCEEDINGS



USENIX, THE UNIX AND ADVANCED COMPUTING SYSTEMS
PROFESSIONAL AND TECHNICAL ASSOCIATION

USENIX Association

Proceedings of the Winter 1993 USENIX Conference

January 25 – 29, 1993
San Diego, California, USA

For additional copies of these proceedings contact

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA

The price is $33 for members and $40 for nonmembers.

Outside the U.S.A and Canada, please add
$25 per copy for postage (via air printed matter).

Past USENIX Technical Conferences

1992 Summer San Antonio	1987 Summer Phoenix
1992 Winter San Francisco	1987 Winter Washington, DC
1991 Summer Nashville	1986 Summer Atlanta
1991 Winter Dallas	1986 Winter Denver
1990 Summer Anaheim	1985 Summer Portland
1990 Winter Washington, DC	1985 Winter Dallas
1989 Summer Baltimore	1984 Summer Salt Lake City
1989 Winter San Diego	1984 Winter Washington, DC
1988 Summer San Francisco	1983 Summer Toronto
1988 Winter Dallas	1983 Winter San Diego

© Copyright 1993 by The USENIX Association
All rights reserved.

ISBN 1-880446-48-0

This volume is published as a collective work.

USENIX acknowledges all trademarks appearing herein, including the following registered and unregistered trademarks:

Holder	Trademark(s)
Adobe Systems, Inc.	PostScript, Display PostScript System
Apple Computer Corp.	Macintosh
Brooktree Corporation	Brooktree, RAMDAC
Chorus Systemes	Chorus
Digital Equipment Corp.	VMS, Ultrix, Decstation, DECsystem, TURBOchannel, DECstation 5000, Alpha
Ingres, Inc.	INGRES
Intel	Paragon
International Business Machines Corp.	AIX, TCF, JFS
Locus Computing Corp	TNC
MIPS Technologies, Inc.	MIPS, R2000, R3000, R4000
Massachusetts Institute of Technology	X Window System
Microsoft Corp.	MS-DOS
Open Software Foundation	OSF/1
SPARC International	SPARC
SUN Microsystems	SunOS, SparcStation, Solaris, SUN
StorageTek, Inc.	StorageTek
TransArc	AFS
UNIX Systems Laboratories, Inc.	UNIX
Xerox Corp.	Xerox
Xerox Corporation	GlobalView

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

TABLE OF CONTENTS

Acknowledgments viii

Preface ix

Author Index x

Plenary Session

Wednesday (9:00-10:20) **Chair: Rob Kolstad**

Opening Remarks and Announcements
Rob Kolstad, BSDI; Dan Geer, Geer Zolot Associates

Keynote Address: Pen-Based Computing and Its Impact
Robert Carr, Go Corporation

Libraries & Links

Wednesday (10:45-12:05) **Chair: Tom Christiansen**

Dictionary and Graph Libraries 1
Stephen C. North & Kiem-Phong Vo, AT&T Bell Laboratories

Linking Shared Segments 13
W. E. Garrett, M. L. Scott, R. Bianchini, L. I. Kontothanassis, R. A. McCallum, J. A. Thomas, R. Wisniewski, & S. Luk, University of Rochester

A Library Implementation of POSIX Threads under UNIX 29
Frank Mueller, Florida State University

New Views

Wednesday (10:45-12:05) **Chair: Peter Honeyman**

Hello World 43
Rob Pike & Ken Thompson, AT&T Bell Laboratories

Es: A shell with higher-order functions 51
Paul Haahr, Adobe Systems Incorporated; Byron Rakitzis, Network Appliance Corporation

Jgraph – A Filter for Plotting Graphs in PostScript 61
James S. Plank, Princeton University

Tuning

Wednesday (1:30-2:50) — Chair: Dinah McNutt

Faster AFS 67
Michael T. Stolarchuk, University of Michigan

The AutoCacher: A File Cache Which Operates at the NFS Level 77
Ronald G. Minnich, Supercomputing Research Center

Pitfalls in Multithreading SVR4 STREAMS and Other Weightless Processes 85
Sunil Saxena, J. Kent Peacock, Fred Yang, Vijaya Verma, Mohan Krishnan, Intel Multiprocessor Consortium

Tools

Wednesday (1:30-2:50) — Chair: Saul G. Wold

WARLOCK – A Static Data Race Analysis Tool 97
Nicholas Sterling, SunSoft, Inc.

DUEL – A Very High-Level Debugging Language 107
Michael Golan & David R. Hanson, Princeton University

The San Diego "Zoo": A multicomputer stress test suite 119
Chris Peak, Locus Computing Corporation, San Diego

Communications

Wednesday (3:30-5:00) — Chair: Dave Taylor

PhoneStation, Moving the Telephone onto the Virtual Desktop 131
Stephen A. Uhler, Bellcore

Glish: A User-Level Software Bus for Loosely-Coupled Distributed Systems 141
Vern Paxson & Chris Saltmarsh, Lawrence Berkeley Laboratory

UNIX Services for Multilevel Storage and Communications Over a Secure LAN 157
Bruno d'Ausbourg & Christel Calas, CERT-ONERA

Xbits

Thursday (9:00-10:20) — Chair: Mary Seabrook

A Sketch Of The Smart Frame Buffer 169
Joel McCormack & Bob McNamara, Digital Equipment Corporation

Wafe – An X Toolkit Based Frontend for Application Programs in Various Programming Languages 181
Gustaf Neumann & Stefan Nusser, Wirtschaftsuniversität Wien

Design and Implementation of a Multi-Threaded Xlib 193
Carl Schmidtmann, Consultant to Digital Equipment Corporation; Michael Tao, Sun Microsystems; Steven Watt, Consultant to Xerox Corporation

Filesystems, I

Thursday (9:00-10:20) **Chair: Dan Geer**

The Design and Implementation of the Inversion File System 205
Michael A. Olson, University of California at Berkeley

Operating System Support for Portable Filesystem Extensions 219
Neil Webber, Epoch Systems, Inc.

File Systems in User Space 229
Paul R. Eggert, Twin Sun, Inc.; D. Stott Parker, UCLA Computer Science Dept.

Overhead

Thursday (10:45-12:05) **Chair: Rob Kolstad**

UNIX Kernel Support for OLTP Performance 241
Hyuck Yoo & Tom Rogers, Sun Microsystems, Inc.

Measurement, Analysis, and Improvement of UDP/IP Throughput for the DECstation 5000 249
Jonathan Kay & Joseph Pasquale, University of California, San Diego

The BSD Packet Filter: A New Architecture for User-level Packet Capture 259
Steven McCanne & Van Jacobson, Lawrence Berkeley Laboratory

I/O, I/O

Thursday (10:45-12:05) **Chair: Jeff Schwab**

The Organization of Networks in Plan 9 271
Dave Presotto & Phil Winterbottom, AT&T Bell Laboratories

Removable Media in Solaris 281
Howard Alt, SunSoft, Incorporated

An Advanced Tape Cataloging System for UNIX Systems 289
Christopher J. Calabrese, AT&T Bell Laboratories

Kernel Improvements

Thursday (1:30-2:50) **Chair: J. Kent Peacock**

Efficient Kernel Memory Allocation on Shared-Memory Multiprocessors 295
Paul E. McKenney & Jack Slingwine, Sequent Computer Systems, Inc.

An Implementation of a Log-Structured File System for UNIX 307
Margo Seltzer, Harvard University; Keith Bostic, University of California, Berkeley; Marshall Kirk McKusick, University of California, Berkeley; Carl Staelin, Hewlett-Packard Laboratories

Exploiting In-Kernel Data Paths to Improve I/O Throughput and CPU Availability 327
Kevin Fall & Joseph Pasquale, University of California, San Diego

Information Discovery

Friday (9:00-10:20) **Chair: Jim Duncan**

Fremont: A System for Discovering Network Characteristics and Problems 335
David C. M. Wood, Sean S. Coleman, & Michael F. Schwartz, University of Colorado

The Enterprise Distributed White-pages Service 349
C. Mic Bowman & Chanda Dharap, Penn. State University

Essence: A Resource Discovery System Based on Semantic File Indexing 361
Darren R. Hardy & Michael F. Schwartz, University of Colorado, Boulder

Monitoring

Friday (9:00-10:20) **Chair: Dick Dunn**

Hardware Profiling of Kernels 375
Andrew McRae, Megadata Pty Ltd.

A Randomized Sampling Clock for CPU Utilization Estimation and Code Profiling 387
Steven McCanne & Chris Torek, Lawrence Berkeley Laboratory

Fault Interpretation: Fine-Grain Monitoring of Page Accesses 395
Daniel R. Edelson, INRIA Project SOR

Filesystems, II

Friday (10:45-12:05) **Chair: Matthew Blaze**

UNIX Disk Access Patterns 405
Chris Ruemmler & John Wilkes, Hewlett-Packard, Inc.

An Analysis of File Migration in a UNIX Supercomputing Environment 421
Ethan L. Miller & Randy H. Katz, University of California, Berkeley

HighLight: Using a Log-structured File System for Tertiary Storage Management 435
John T. Kohl, University of California, Berkeley and Digital Equipment Corporation; Carl Staelin, Hewlett-Packard Laboratories; Michael Stonebraker, University of California, Berkeley

O/S Implementations

Friday (10:45-12:05) **Chair: Steve McDowell**

An OSF/1 UNIX for Massively Parallel Multicomputers 449
Roman Zajcew, Paul Roy, David Black, Chris Peak, Paulo Guedes, Bradford Kemp, John LoVerso, Michael Leibensperger, Michael Barnett, Faramarz Rabii, & Durriya Netterwala, OSF Research Institute and Locus Computing Corporation

An Implementation of UNIX on an Object-oriented Operating System 469
Yousef A. Khalidi & Michael N. Nelson, Sun Microsystems Laboratories, Inc.

The Nachos Instructional Operating System 481
Wayne A. Christopher, Steven J. Procter, & Thomas E. Anderson, University of California at Berkeley

Cache & Carry

Friday (1:30-2:50) **Chair: David S. H. Rosenthal**

The Design and Implementation of a Mobile Internetworking Architecture 489
John Ioannidis & Gerald Q. Maguire, Jr., Columbia University

Mobile Computing Environment Based on Internet Packet Forwarding 503
Hiromi Wada, Takashi Yozawa, Tatsuya Ohnishi, & Yasunori Tanaka, Matsushita Electric Industrial Co., Ltd.

The Compression Cache: Using On-line Compression to Extend Physical Memory 519
Fred Douglis, Matsushita Information Technology Laboratory

ACKNOWLEDGMENTS

GENERAL CHAIR

Rob Kolstad, *BSDI*

PROGRAM CHAIR

Dan Geer, *Geer Zolot Associates*

PROGRAM COMMITTEE

Matthew Blaze, *AT&T Bell Laboratories*
Tom Christiansen, *CONVEX Computer Corp.*
Clement T. Cole, *Locus Computing Corp.*
James Duncan, *Penn State University*
Dick Dunn, *Eklektix*
Dan Geer, *Geer Zolot Associates*
Peter Honeyman, *The University of Michigan*
Dan Klein, *LoneWolf*
Rob Kolstad, *BSDI*
Steve McDowell, *EXLOG*
Dinah McNutt, *Tivoli Systems*
Kent Peacock, *Intel*
Gretchen Phillips, *State University of New York*
David Rosenthal, *Sun Microsystems*
Jeffrey R. Schwab, *Purdue University*
Mary Seabrook, *Open Systems Solutions*
Dave Taylor, *SunWorld Magazine*
Saul G. Wold, *Sun Microsystems*

READERS

Jaap Akkerhuis, *AT&T Bell Laboratories*
Matthew A. Bishop, *Dartmouth College*
R. Lee Damon, *IBM*
Judith E. Grass, *AT&T Bell Laboratories*
Bruce Keith, *Digital Equipment Corp.*
John Kohl, *University of California, Berkeley*
Jeffrey Mogul, *Digital Equipment Corp.*
Mike Olson, *University of California, Berkeley*
Bjorn Satdeva, */sys/admin*
Kevin Smallwood, *Purdue University*
Jon Tankersley, *Amoco Production Company*
G. Win Treese, *Digital Equipment Corp.*
Bernhard Wagner, *Ciba-Geigy AG*
Mark Weiser, *Xerox*
Peter Zadrozny, *Sun Microsystems de Mexico*

INVITED TALKS COORDINATORS

Tom Cargill, *Consultant*
Bob Gray, *US WEST Advanced Technologies*

BOF COORDINATOR

Judy DesHarnais, *USENIX Association*

TUTORIAL COORDINATOR

Daniel V. Klein, *USENIX Association*

WORK-IN-PROGRESS COORDINATOR

Lisa A. Bloch

TERMINAL ROOM COORDINATOR

Gretchen Phillips, *SUNY Buffalo*

PROGRAM COMMITEE SCRIBE

Rob Kolstad, *BSDI*

PROCEEDINGS PRODUCTION

Rob Kolstad, *BSDI*
Carolyn Carr, *USENIX*
Malloy Lithographing, Inc.

USENIX MEETING PLANNER

Judith F. Desharnais, *USENIX Association*

USENIX EXECUTIVE DIRECTOR

Ellie Young, *USENIX Association*

USENIX SUPPORT STAFF

Marilynn Allemann, *USENIX Association*
Diane DeMartini, *USENIX Association*
Patrick Dunne, *USENIX Association*
Andrea Galleni, *USENIX Association*
Toni Veglia, *USENIX Association*

VENDOR DISPLAY COORDINATOR

Cynthia Deno, *USENIX Consultant*

CONFERENCE PROMOTION

Eva Bernstein, *USENIX Consultant*
Cynthia Deno, *USENIX Consultant*

LOCAL ARRANGEMENTS

David Wollner, *Accelerated Systems, Inc.*

PREFACE

Rob Kolstad writes:

San Diego '93 marks the trial of some new ideas for the USENIX association. For the first time, a General Chair coordinates the activities of program chair, invited talks coordinators, and a host of other conference details. This conference sports three parallel tracks, each filled with a cornucopia of exciting technical information. My goal has been to provide "something for everybody" – a wide variety of technical presentations with very high quality.

Conference coordination business is dramatically eased by a large number of volunteers and the wonderful staff at USENIX. The thing that amazed me was the staff's ability to call and say, "You haven't forgotten the call for papers needs the [whatever] filled in by next Tuesday, have you?". I was able to feel competent and dignified in my forgetfulness and still fulfill the requirements of the job. The entire support group (authors, coordinators, committees, and readers alike) came through to enable all deadlines to be met and this conference to happen on the usual short publication schedule that it requires. Thanks to all!

As UNIX and our industry suffer attacks from many sides (other operating systems, blitzkriegs of marketing from competing industries, intellectual property challenges from patents through actual lawsuits), it is interesting to watch the technical side of the industry continue to mature and grow, to cross-pollinate, and in general to continue to explore alternatives to find a suite of "right answers" for now and for the future. I'm sure you will find many new "right answers" in these proceedings.

At a recent trade show, I was discussing career planning and predicting the future with a fellow engineer about my age. We agreed that neither of us could have predicted the turns our careers have made nor the interesting technical and non-technical aspects that our industry has encountered over the last two decades. We pretty much felt that even our optimistic predictions could not have imagined the potpourri of fine technical tools that we see presented in proceedings like this one. Surely it is a great time to be a member of our community.

Please accept my welcome to the Winter 1993 USENIX conference – especially if you have not attended before. I hope you will accomplish your goals in attending and will return (with others). When you have ideas for improvements and enhancements, please let me (or any USENIX staffer or director) know so that we can continue to raise our conference's quality.

Thanks for coming!

Rob Kolstad
General Chair

Dan Geer writes:

It has been my pleasure to serve as Technical Program Chair for this Conference and to work with both an energetic and smart group of committee members and readers. All of us want to thank submitters, accepted and not, for showing us their ideas. The strength of USENIX is the strength of its technical content, and that entirely flows from the works of authors and reviewers. Both are necessary; neither is sufficient. I hope each attendee will seek out program committee members, readers, and authors and thank them for the essential service they provide us all. After you have done that, make plans to submit your own work – we will all thank you for it.

Dan Geer
Program Chair

AUTHOR INDEX

Howard Alt 281
Thomas E. Anderson 481
Bruno d'Ausbourg 157
Michael Barnett 449
R. Bianchini 13
David Black 449
Keith Bostic 307
C. Mic Bowman 349
Christopher J. Calabrese 289
Christel Calas 157
Wayne A. Christopher 481
Sean S. Coleman 335
Chanda Dharap 349
Fred Douglis 519
Daniel R. Edelson 395
Paul R. Eggert 229
Kevin Fall 327
W. E. Garrett 13
Michael Golan 107
Paulo Guedes 449
Paul Haahr 51
David R. Hanson 107
Darren R. Hardy 361
John Ioannidis 489
Van Jacobson 259
Randy H. Katz 421
Jonathan Kay 249
Bradford Kemp 449
Yousef A. Khalidi 469
John T. Kohl 435
L. I. Kontothanassis 13
Mohan Krishnan 85
Michael Leibensperger 449
John LoVerso 449
S. Luk 13
Gerald Q. Maguire, Jr. 489
R. A. McCallum 13
Steven McCanne 259
Steven McCanne 387
Joel McCormack 169
Paul E. McKenney 295
Marshall Kirk McKusick 307
Bob McNamara 169
Andrew McRae 375
Ethan L. Miller 421
Ronald G. Minnich 77
Frank Mueller 29
Michael N. Nelson 469
Durriya Netterwala 449
Gustaf Neumann 181
Stephen C. North 1
Stefan Nusser 181
Tatsuya Ohnishi 503
Michael A. Olson 205
D. Stott Parker 229
Joseph Pasquale 249
Joseph Pasquale 327
Vern Paxson 141
J. Kent Peacock 85
Chris Peak 119
Chris Peak 449
Rob Pike 43
James S. Plank 61
Dave Presotto 271
Steven J. Procter 481
Faramarz Rabii 449
Byron Rakitzis 51
Tom Rogers 241
Paul Roy 449
Chris Ruemmler 405
Chris Saltmarsh 141
Sunil Saxena 85
Carl Schmidtmann 193
Michael F. Schwartz 335
Michael F. Schwartz 361
M. L. Scott 13
Margo Seltzer 307
Jack Slingwine 295
Carl Staelin 307
Carl Staelin 435
Nicholas Sterling 97
Michael T. Stolarchuk 67
Michael Stonebraker 435
Yasunori Tanaka 503
Michael Tao 193
J. A. Thomas 13
Ken Thompson 43
Chris Torek 387
Stephen A. Uhler 131
Vijaya Verma 85
Kiem-Phong Vo 1
Hiromi Wada 503
Steven Watt 193
Neil Webber 219
John Wilkes 405
Phil Winterbottom 271
R. Wisniewski 13
David C. M. Wood 335
Fred Yang 85
Hyuck Yoo 241
Takashi Yozawa 503
Roman Zajcew 449

Dictionary and Graph Libraries

Stephen C. North & Kiem-Phong Vo – AT&T Bell Laboratories

ABSTRACT

Searching and graph algorithms are pervasive in computer programs. We describe libraries for dictionaries and graphs that offer efficient implementations with more flexibility and generality than hand-crafted algorithms. *libdict* maintains ordered and unordered dictionaries, under a common interface. *libgraph* supports operations on attributed graphs, including reading and writing graph files as a basis for creating graph processing tools.

Introduction

Searching and graph algorithms are pervasive in computer programs. As these techniques are abstract, programmers often hand-craft algorithms to match the type of objects being manipulated. At best, this causes duplicated code. At worst, algorithms may be chosen for the wrong reasons and then implemented badly, resulting in inefficient or even incorrect applications. This speaks for the need of libraries that are well-designed and well-implemented. Toward this end, we shall describe two libraries for object storage and graph manipulation: *libdict* and *libgraph*.

libdict provides functions to manage run-time dictionaries. Each dictionary contains objects that may have some implicit ordering or may be unordered. The main contribution of *libdict* is the use of efficient adaptive data structures to support both ordered and unordered dictionaries. Self-adjusting binary trees [ST] are used for ordered dictionaries, while hash tables with self-adjusting chains are used for unordered ones. Both of these data structures have good theoretical performance. We shall give evidence that they also perform well in practice when compared with other popular packages that provide similar functions.

libgraph manages both run-time and file representation of graphs. It provides convenient functions to build graphs and subgraphs whose nodes and edges may have application-dependent attributes. The external file representation of graphs makes it simple to write graph processing tools or filters in the traditional UNIX programming style. *libgraph* uses *libdict* for run-time graph object storage.

The combination of these libraries makes it easy to write applications that perform actions ranging from simple sort and search to manipulation of sophisticated graph structures. We have written a number of graph layout tools based on the libraries such as *dot* [GNV, KN] for directed graphs; *neato* for undirected graphs; *tred*, a transitive reduction filter that removes edges that are redundant with respect to reachability; and *scc*, that finds strongly connected components. The external graph file format makes it easy to share data among the various programs that underlie these layout tools.

libdict

Why libdict

The problem addressed by *libdict* is to store and search objects. There are many proposed algorithms and associated data structures for this problem [Knu,Sed]. Regardless of the method, the programmer interface usually contains primitives to insert, delete and find objects. If objects are ordered, some type of binary trees is often used to store them. With the right type of trees, each primitive function consumes on the average $O(\log n)$ time where n is the total number of objects. This is optimal because just sorting the objects requires $O(n \log n)$ time. On the other hand, if the order of objects is not important, it is well-known that hashing takes $O(1)$ average time per primitive operation.

The availability of many algorithms for essentially the same purpose but with different performance levels causes a problem for implementors of general purpose libraries. The temptation is great for creating separate packages tailored to different types of applications. Indeed, this is the case on various C and C++ platforms.

On some flavor of UNIX systems [SV], ordered dictionaries are handled by *tsearch()*, a tree-based package, while unordered dictionaries are handled by *hsearch()*, which maintains a hash table. These packages employ completely distinct interfaces so that it is not easy to take advantage of their services in applications that require simultaneous manipulation of both ordered and unordered objects. *hsearch* also imposes a severe limitation by allowing only one hash table per application.

In the C++ environment, the *map* class library [Koe] by Andy Koenig is by far the most popular package for on-line dictionaries. *map* requires that the objects be ordered in some way, thus sacrificing efficiency when objects are unordered and could be hashed instead.

libdict solves the above software engineering problems by providing a uniform interface for

manipulating dictionaries. Multiple dictionaries can be created. Each dictionary can be ordered or hashed. In fact, the ordering function of a dictionary can be dynamically changed at any point during its lifetime.

When a dictionary is ordered, *libdict* uses a self-adjusting binary tree or splay tree to store the objects. Splay tree technology is beyond the scope of this paper; see Sleator and Tarjan's paper [ST] for details. It is sufficient to understand that using splay trees, the space overhead for each stored object can be limited to two pointers. Each primitive dictionary operation takes O(log*n*) time in an amortized sense. Amortization means that the cost of an operation is taken as the average over the entire sequence of insert/delete/search operations to build and manipulate a dictionary, although a particular operation may take longer, even O(*n*) in the worst case. This is because splay trees are adaptive data structures that rearrange themselves based on search distribution. In particular, splay trees adapt well to biased search patterns in which certain subsets are searched more frequently than others. In such cases, frequently searched objects percolate to the top of the tree so that finding them is fast. It is also known that traversing the objects stored in a splay tree in order takes linear time.

When a dictionary is unordered, *libdict* uses a hash table with self-adjusting chains to store the objects. That is, objects that hash to the same bucket are stored in a chain. When an object is accessed, it is moved to the front of its chain. Using this data structure, the search time per primitive operation is constant time on the average assuming that we have a good hash function that distributes objects evenly. The space overhead is about 2.5 pointers per object.

For efficiency comparison, we ran a set of benchmark tests of *libdict*, *map*, *tsearch* and *hsearch*. Each test run consists of inserting into a dictionary all integers in the half-open range *[0,n)*, then walking through the dictionary exactly once. All tests are done on a Solbourne (SPARC) server running SUN OS4.1.1. To reduce variance in timings, the tests were done at night with the system quiescent, and the resulting times are averaged over three consecutive runs. In the following tables, *dict-tree* stands for the use of *libdict* when objects are ordered, *dict-hash* for *libdict* when objects are in hash mode. The time data are shown in seconds while space in units of K-bytes.

Table 1 shows the timings of different packages for *n* varying from *1000* to *10000*. In this case, the integers are inserted in their natural order. Note that for *dict-hash* and *hsearch()*, the ordering is ignored by the respective packages. The data show clearly that both *tsearch()* and *hsearch()* exhibit a disturbing quadratic time behavior. For *tsearch()*, this is because its binary tree reduces to a linear linked list when elements are inserted in their order. We are not familiar enough with the implementation of *hsearch()* to tell why it behaves badly. However, we note that the timing result for *hsearch()* is somewhat unfair because *hsearch()* hashes objects by their key strings so we have to include the time taken to convert integers from their binary to text representation. The C++ *map* package outperforms both *tsearch()* and *hsearch()* but it runs about twice as slow as *dict-tree*. As expected, *dict-hash* performs best.

Table 2 shows timing results when the integers are inserted in random order. Now *tsearch()* is competitive with *dict-tree* because the random order insertions assure that the tree will be approximately balanced. There is no essential performance difference for both C++ *map* and *dict-hash*. This is to be

Method	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
dict-tree	0.03	0.10	0.14	0.21	0.24	0.32	0.37	0.43	0.49	0.55
tsearch	1.61	6.82	15.07	27.94	50.38	72.48	97.84	131.91	161.48	206.73
C++ map	0.06	0.17	0.30	0.39	0.51	0.61	0.73	0.84	0.96	1.12
dict-hash	0.03	0.08	0.12	0.17	0.22	0.27	0.31	0.33	0.42	0.45
hsearch	0.65	1.67	2.70	7.15	8.20	13.13	20.38	32.93	28.60	36.92

Table 1: Time usage (in seconds) when inserted in order

Method	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
dict-tree	0.05	0.12	0.21	0.29	0.35	0.46	0.49	0.60	0.70	0.76
tsearch	0.05	0.12	0.21	0.27	0.35	0.43	0.51	0.62	0.67	0.76
C++ map	0.08	0.16	0.28	0.36	0.45	0.59	0.68	0.78	0.94	1.02
dict-hash	0.03	0.08	0.14	0.17	0.22	0.27	0.31	0.36	0.41	0.47
hsearch	0.60	1.37	2.49	6.15	7.64	12.81	19.34	29.47	27.15	35.24

Table 2: Time usage (in seconds) when inserted randomly

expected since both data structures are impervious to object ordering (think!). The interesting part is that *dict-tree* is much faster when objects were inserted in order than when they were inserted randomly. This is because the splay tree structure takes advantage of the former case and reduces the work load. In any case, *dict-tree* and *dict-hash* outperform the other packages.

Finally, Table 3 shows the space consumption by the various packages. As expected, *dict-tree* and *tsearch()* use the same amount of space since both use simple binary trees with two child links per node to store objects. *dict-hash* uses a little more space due to the hash table. The poor space measure for *hsearch* is again somewhat unfair because integers have to be converted to text form to create keys. Somewhat more disturbing is the C++ *map* package which consumes the most amount of memory (after factoring out the space used to store text in *hsearch()*.)

Libdict functions

Libdict is designed for flexibility and efficiency. Therefore, there is a wide range of functions and macros that allow applications to manipulate objects at a high level or even at the internal representation level. A full description of all functions in *libdict* is beyond this paper. Below we describe the main functions in *libdict*.

Dict_t dopen(uchar* (*makef)(), void (*freef)(), int (*comparf), ulong (*hashf)())*

dclose(Dict_t dict)*

These functions create and close dictionaries. *(*makef)(uchar* obj)* creates a new object from the prototype *obj*. *freef(obj)* deletes the object *obj*. *comparf(obj1,obj2)* compares two objects and returns a value that is *<0*, *=0*, or *>0* to indicate whether *obj1* is smaller, equal to, or larger than *obj2*. When the dictionary is hashed, the return value of *(*comparf)()* is interpreted as a boolean. Finally, *(*hashf)(obj)*, if given, computes the hash values of objects when the dictionary is in hash mode.

dhash(Dict_t dict, int size)*

This function enables hash mode for *dict* if *size>=0*. Otherwise, *dhash()* restores *dict* to order mode. If *size==0*, the number of slots in the hash table is dynamically adjusted by *libdict*. If *size>0*, the hash table is fixed at this size.

dview(Dict_t dict, Dict_t* viewdict)*

This function sets a view path from the dictionary *dict* to the dictionary*viewdict*. This means that a search for an object in *dict* or a walk through it will continue to *viewdict* and any dictionaries recursively viewed thereof. A view can be terminated by specifying *NULL* for *viewdict*. View pathing is useful for programs that manipulate objects in different but related dictionaries. For example, in a parser, local variables in different scopes may be stored in different dictionaries. If the dictionaries in nested scopes are connected by a view path, the search for a variable can begin at the current scope and continue upward through enclosing scopes.

dreorder(Dict_t dict, int (*comparf)())*

This function changes the ordering of objects in *dict* to a new one defined by the function *(*comparf)()*. The same is done for all dictionaries that are on a view path from *dict* (see *dview()*). In each reordered dictionary, newly duplicated objects are eliminated.

dinsert(Dict_t dict, uchar* obj)*
ddelete(Dict_t dict, uchar* obj)*

These function insert or delete objects. For *dinsert()*, if *obj* is already in the dictionary, it is not reinserted.

dsearch(Dict_t dict, uchar* obj)*

This function returns an object in the dictionary that matches the prototype object *obj*, or *NULL* for no match.

dfirst(Dict_t dict)*
dlast(Dict_t dict)*

These functions return the first/last object in the dictionary. If the dictionary is in hash mode, an internal order determines which object is first/last.

dnext(Dict_t dict, uchar* obj)*
dprev(Dict_t dict, uchar* obj)*

These function return the object following/preceding *obj* as defined by the dictionary ordering. If the dictionary is in hash mode, the dictionary ordering is not well-defined and may change dynamically when calls to *dsearch()* and *dinsert()* are made. The standard ways to walk a dictionary are:

for(obj = dfirst(dict); obj != NULL; obj = dnext(dict,obj))

or *for(obj = dlast(dict); obj != NULL; obj = dprev(dict,obj))*

Method	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
dict-tree	32	56	80	104	128	152	176	200	224	248
tsearch	32	56	80	104	128	152	176	200	224	248
C++ map	40	72	104	136	168	200	232	264	296	328
dict-hash	40	64	88	112	144	168	192	216	264	280
hsearch	48	72	112	128	184	200	224	240	328	344

Table 3: Space usage (in K-bytes)

Note that if *dict* has a view on some other dictionaries (see *dview()*), the loop will also traverse these dictionaries. In this case, only one such *for(;;)* loop is allowed for *dict*. Nested loops may result in unexpected behavior.

Using libdict

This section shows the use of *libdict* via a simple exercise: to read a set of words from a file, eliminate any duplications, then emit the unique words in the alphabetic order. The alphabetic order of words is defined as in a dictionary where the upper case version of a letter is considered smaller than its lower case version. We shall assume that the words are given in the standard input stream, one per line. The output list of words will be written to the standard output stream, one per line. All IO operations use the *sfio* package[KV]. For simplicity, we shall omit error checking.

Figure 1 shows the main processing code of the program. Using the line numbers, below is the description of various program parts.

- `1-2`: The header files `sfio.h` and `dict.h` contains declarations of the types and functions required by the program. For example, `sfio.h` defines the standard input and output streams `sfstdin` and `sfstdout`.
- `3-5`: These lines declare the functions required to manipulate a dictionary. `newword()` is a function to create a word given a copy of it. This is needed because the function `sfgetr()` on line 14 returns a word in some internal *sfio* buffer area that can be overwritten in a future call. The standard C string comparison function `strcmp()` is used for fast detection of duplications. `cmpalpha()` is a more complex comparison function to compare strings by their alphabetic order. `hashword()` computes the hash value of a word.
- `9-10`: These lines open a dictionary and set it in hash mode. Hash mode is used here so that each word insertion takes constant time on average. Note that to save computing time we use `strcmp()` to eliminate duplications instead of `cmpalpha()`.
- `11-12`: These lines read and insert words into the dictionary. If a word is already in the dictionary, `dinsert()` will not insert it again.
- `13-14`: All the words have been read and duplications have been eliminated. The call `dreorder(dict,cmpalpha)` changes the word ordering to an alphabetic ordering as defined by `cmpalpha()`. The call `dhash(dict,-1)` turns off hashing so that the words will be in their defined order.
- `16-17`: These lines walk through the dictionary and emit the words in alphabetic order.

Figure 2 shows the code for `newword()`, `hashword()` and `cmpalpha()`. The function `hashword()` uses the macro function `dstrhash()` provided by *libdict* to compute a hash value for a given string. `cmpalpha()` compares two words by their alphabetic order and returns a value that is `<0`, `0` or `>0` accordingly as the first word is considered smaller, equal or larger than the second word.

```
 1 #include            <sfio.h>
 2 #include            <dict.h>
 3 extern Dmake_f      newword;
 4 extern Dcompar_f    strcmp, cmpalpha;
 5 extern Dhash_f      hashword;
 6 main()
 7 {   char*   word;
 8      Dict_t* dict;
 9     dict = dopen(newword,(Dfree_f)0,strcmp,hashword);
10     dhash(dict,0);
11     while((word = sfgetr(sfstdin,'\n',1)) != (char*)0)
12         dinsert(dict,word);
13     dreorder(dict,cmpalpha);
14     dhash(dict,-1);
16     for(word = (char*)dfirst(dict); word; word = (char*)dnext(dict,word))
17         sfputr(sfstdout,word,'\n');
18 }
```

Figure 1: Program to uniq-ize words

To recap, the above solution to make an alphabetically ordered list of words highlights a few main features of *libdict*. First, whether a dictionary is ordered or unordered, the programmer interface is the same. Moreover, switching from one mode to the other is only a matter of a function call. This allows the programmer to tune for performance using the right data structure at the right time in the program. Second, it is often the case in programming that a collection of objects must be viewed in different ways depending on the situation in which they are used. *libdict* simplifies doing this by allowing arbitrary changes of the comparison function. Finally, the output loop of the program shows how the objects of a dictionary can be walked using an ordinary `for(;;)` loop and not requiring the construction of a call back function common to packages such as *tsearch()*. This is especially nice for applications in which the processing of an object may depend on some state values. In such cases, the states can be kept local to the loop and need not be maintained in some external or static variables across different invocations of the call back function.

libgraph

libgraph supports operations on directed or undirected attributed graphs. These operations include reading and writing graph files. Attributes are bound dynamically, not pre-defined for a specific application. Thus graph files are compatible across all programs that use *libgraph*, providing a standard file format for graph databases and tools. This also makes it convenient to run graph filters in pipelines. Another interesting aspect of *libgraph* is a way of defining recursive subgraphs for encoding structural information in graphs.

```
 1 unsigned char* newword(unsigned char* word)
 2 {   unsigned char* w;
 3     w = (unsigned char*) malloc(strlen((char*)word)+1);
 4     strcpy((char*)w,(char*)word);
 5     return w;
 6 }
 7
 8 unsigned long hashword(unsigned char* word)
 9 {   unsigned long h;
10     dstrhash(h,word,-1);
11     return h;
12 }
13 cmpalpha(unsigned char* s1, unsigned char* s2)
14 {   int c1, c2;
15     while((c1 = *s1++) != 0)
16     {    if((c2 = *s2++) == 0)
17             return 1;
18          if(c1 >= 'A' && c1 <= 'Z')
19          {   if(c2 >= 'a' && c2 <= 'z')
20              {   c2 = 'A' + (c2 - 'a');
21                  return c1 <= c2 ? -1 : 1;
22              }
23          }
24          else if(c1 >= 'a' && c1 <= 'z')
25          {   if(c2 >= 'A' && c2 <= 'Z')
26              {   c2 = 'a' + (c2 - 'A');
27                  return c1 >= c2 ? 1 : -1;
28              }
29          }
30          if((c1 -= c2) != 0)
31              return c1;
32     }
33     return *s2 ? -1 : 0;
34 }
```

Figure 2: Copying, hashing, comparing words

On the other hand, *libgraph* deals only with graph representation. Common graph algorithms such as depth-first search or finding strongly connected components are representation independent and could be written on top of *libgraph*.

Graph Model and Runtime Representation

Graphs are sets of attributed nodes, edges, and subgraphs. Subgraphs may contain any graph elements, including other subgraphs. When a node or edge is inserted into a subgraph, it is also inserted into all superior graphs as necessary. (While these semantics arise naturally from a set-theoretic view of subgraphs, but their reliance on side-effects raises some questions about porting *libgraph* to an applicative language such as SML.) Subgraphs may be passed to almost all *libgraph* functions that expect a graph pointer. This makes it natural to use subgraphs in functions that create, filter, or operate on graphs.

Conventionally, graphs are represented at runtime as adjacency matrices or edge lists. Adjacency matrices have unacceptable space overhead for our intended application. Edge lists, on the other hand, are complicated by our model of subgraphs, since a node can have has a different edge list in every subgraph to which it belongs. Further, we would like subgraphs to be inexpensive to create on the fly, that is to say, $O(1)$ time, not $O(|V|)$.

Our approach uses *libdict* to store sets of nodes and edges. Each main graph or subgraph has a dictionary of nodes indexed by an internal node number, and dictionaries of edges indexed as both in- and out-edges. The main advantage of using *libdict* here is that it yields log time random probes, and amortized linear time traversal of node and edge lists. That is, while finding an individual node or edge takes $O(\log(|V|))$ or $O(\log(|E|))$ time, sequentially moving to the next item takes only constant time. This is important because visiting nodes or edges in this way are common operations and we want them to be efficient. Further, the input order of nodes and edges can be preserved. Using *libdict* also has secondary benefits. We use it for other dictionaries behind the scenes in *libgraph* that store reference-counted strings and attribute symbol tables.

A secondary benefit is that the dictionary reordering feature of *libdict* allows programmer-defined ordering of edges adjacent to a given node. For example, if nodes have geometric coordinates, a programmer can define a comparison function for clockwise edge ordering.

Functions for Graphs and Subgraphs

Libgraph has several primitives; see Figure 3.

This is an abstracted version of the graph struct and related functions. When a new graph is created, its kind is given as DIRECTED, STRICT_DIRECTED, UNDIRECTED, or STRICT_UNDIRECTED. Strict graphs may not have multiple or self-edges, as assumed in many graph algorithms. The graph name is advisory information available to a *libgraph* application, possibly to keep a list of graphs that have been loaded. New nodes and edges of a graph or subgraph are constructed using attributes in template nodes and edges kept in `proto->n` and `proto->e`.

```
typedef struct graph_t {
    char            *name,kind;
    graphdata_t     *univ;
    Dict_t          *nodes,*inedges,
                    *outedges;
    graph_t         *root;
    node_t          *meta_node;
    proto_t         *proto;
    graphinfo_t      u;
} graph_t;

void          initgraphs();
graph_t      *newgraph (char *name, int graph_type);
graph_t      *newsubg (graph_t *g, char *name);
graph_t      *getsubg (graph_t *g, char *name);
void          putsubg (graph_t *g, graph_t *subg);
void          delgraph (graph_t *g);
int           n_nodes (graph_t *g), n_edges (graph_t *g);
int           contains (graph_t *g, void *obj);

node_t       *metanode_of (graph_t *g);
graph_t      *realgraph_of (node_t *metanode);
```

Figure 3: *Libgraph* Primitives

Other primitives in the above section deal with membership of subgraphs, or individual nodes and edges within subgraphs. Because we need some way to traverse the subgraph hierarchy, it is represented as an auxiliary directed graph associated with every "main" graph. The auxiliary graph nodes and edges may be searched using *libgraph* primitives, and there are two additional functions (`metanode_of` and `realgraph_of`) that map between auxiliary graph nodes and their subgraphs.

Functions for Nodes and Edges

Figure 4 describes the functions that deal with nodes and edges. A node is identified either by name, or by an internal number for faster searches. An edge is identified by its endpoints (for multi-edges there is also a key), or by internal number. Edges may have port identifiers, but they are simply maintained as string values; their interpretation is up to application programs.

The usual way of visiting the nodes of a graph is:

for (v = firstnode(g); v; v = nextnode(g,v))
visit_node(v);

Edges of a node v may be visited by:

for (e = firstedge(g,v); e; e = nextedge(g,e,v))
visit_edge(e);

In a directed graph, it is common to traverse in- or out-edges.

for (e = firstout(g,v); e; e = nextout(g,e))
visit_edge(e);

Functions for Attributes

Figure 4 shows the functions that deal with attributes. The elements (nodes, edges, or subgraphs) of a given graph have the same attribute names. When a new attribute is created, its default value are given. If the graph pointer is non-NULL, the graph's elements are updated to contain the new value. If a NULL graph pointer is passed, the declaration is remembered and will be applied to all graphs that are created in the future, including graphs read from files. This is useful for programs to pre-define and pre-allocate certain attributes for all graphs they process. The other primitives in the above set are for reading and writing attributes. Values may be referenced by attribute name, or by a more efficient internal index. As previously

```
typedef struct node_t {
    char                *name;
    int                 id;
    graph_t             *graph;
    nodeinfo_t          u;
} node_t;
typedef struct edge_t {
    int                 id;
    node_t              *head,*tail;
    char                *key;
    char                *hport,*tport;
    edgeinfo_t          u;
} edge_t;
node_t              *newnode (graph_t *g, char *name);
node_t              *getnode (graph_t *g, char *name);
node_t              *firstnode (graph_t *g);
node_t              *nextnode (graph_t *g, node_t *n);
void                putnode (graph_t *g, node_t *n);
void                delnode (graph_t *g, node_t *n);

edge_t              *newedge (graph_t *g, node_t *u, node_t *v);
edge_t              *getedge (graph_t *g, node_t *u, node_t *v);
edge_t              *firstedge (graph_t *g, node_t *n);
edge_t              *nextedge (graph_t *g, edge_t *e, node_t *v);
edge_t              *firstout (graph_t *g, node_t *n);
edge_t              *nextout (graph_t *g, edge_t *e);
edge_t              *firstin (graph_t *g, node_t *n);
edge_t              *nextin (graph_t *g, edge_t *e);
void                putedge (graph_t *g, edge_t *e);
void                deledge (graph_t *g, edge_t *e);
```

Figure 3: Node and Edge Functions

mentioned, the edge attribute `key` has special treatment for distinguishing multi-edges.

Because *libgraph* and its caller pass character strings to each other, there is a question as to where these strings are allocated and freed. A straightforward policy is that *libgraph* and its client are each responsible for their own memory management. For example, when a program passes a string to `graph_setval`, *libgraph* makes a copy; likewise `graph_getval` returns a string, the caller must account for the possibility of the string being freed and overwritten if attributes are subsequently edited.

Graph Files

*graph_t *read_graph (FILE *infile);*

*int write_graph (graph_t *g, FILE *outfile);*

These functions invoke the graph file parser or printer. The file language is illustrated by some examples below. In designing the file language, the important characteristics were that files not only correctly record a graph's runtime state, but that graph files be convenient for humans to read and edit. Figure 3 lists example files of directed and undirected graphs. Our file language has similarities to the one used in the graph editor EDGE [New], though an important advantage of the *libgraph* version is that it is general-purpose, without hard-wired attributes.

To illustrate how several of these functions are combined in a complete program, the following example is a filter that sets the color of all red nodes to blue.

Several reasonable questions arise concerning with the graph language. Why not make a simpler format using tab-separated fields? Such formats are too rigid and make it difficult to edit graphs, such as

```
typedef struct attrsym_t {
    char                *name,*value;
    int                 index;
} attrsym_t;
attrsym_t*          new_globattr (graph_t *g, char *name, char *value);
attrsym_t*          new_nodeattr (graph_t *g, char *name, char *value);
attrsym_t*          new_edgeattr (graph_t *g, char *name, char *value);
attrsym_t*          get_attrdcl (void *obj, char *name);

char                *graph_getval (void *obj, char *name);
int                 graph_setval (void *obj, char *name, char *value);
int                 graph_indexof (void *obj, char *name);
char                *graph_igetval (void *obj, int index);
int                 graph_isetval (void *obj, int index, char *value);
```

Figure 4: Attribute Functions

```
#include <graph.h>
main()
{
    graph_t     *g;
    attrsym_t   *a;

    while (g = read_graph(stdin)) {
        a = get_attrdcl(g->proto->n,"color");
        if (a == NULL)
            fprintf(stderr,"graph %s doesn't have node colors\n",g->name);
        else {
            for (n = firstnode(g); n; n = nextnode(g,n))
                if (!strcmp(graph_getval(n,"color"),"red"))
                    graph_setval(n,"color","blue");
        }
        write_graph(g,stdout);
    }
    exit(0);
}
```

Figure 5: Filter to Color All Red Nodes

to add new attributes. Why not use a data language such as G2 or IDL [NNGS] to store graphs? Such data languages are lower-level, and thus not convenient for our applications. Since graph files may be created by non-technical users, or by simple shell or awk scripts, the format should be flexible yet straightforward. It would be quite awkward for users to encode *libgraph*'s subgraphs in terms of low-level data pointers. Another question is, how is one expected to express concepts such as layout constraints in the graph language? Though the language has no high-level commands or constraints, these can sometimes be simulated by tagging appropriate attributes on nodes or subgraphs that contain sets of nodes or edges of interest. In *dot*, a group of nodes can be kept on the same rank or made into a cluster using subgraphs this way. Admittedly, in the future we would like to extend the language to incorporate constraints as advisory information to applications.

```
digraph sample_1 {
    node [shape=box];
    a -> b -> c;
    c -> {x y z};
    subgraph top {
        node [shape = circle];
        label = "hello world";
        n0 n1 n2;
    }
    /* creates edges from z to
                        n0,n1,n2 */
    z -> subgraph top;
}

graph G {
    run -- intr;
    intr -- runbl;
    runbl -- run;
    run -- kernel [len=2.0,w=10];
    kernel -- zombie;
    kernel -- sleep;
    kernel -- runmem;
    sleep -- swap;
    swap -- runswap;
    runswap -- new;
    runswap -- runmem;
    new -- runmem;
    sleep -- runmem;
}
```

Programmer-defined Fields

Character string attributes, while useful for file I/O, are not adequate for all purposes. In arithmetic operations, not only does the cost of string conversion dominate computation, but *libgraph* function call notation seems inconvenient when compared to C code that directly names struct members. To solve this, programmers can incorporate application-specific data in the u fields of graph, node, and edge structs. For example, a program might need a mark on every node:

```
typedef struct nodeinfo {
    unsigned char mark;
} nodeinfo;
```

A programmer could then write

```
for (n = nodelist(g); n;
                n = nextnode(g,n)) {
    if (n->u.mark == FALSE) dfs(n);
}
```

Though there is no automatic conversion between programmer-defined internal values and external attributes, it is usually easy to write short procedures to do this explicitly when a graph is read or written. This is admittedly low-tech, but simpler and more understandable than automatically generating graph libraries parameterized for each application.

Experience

libgraph is the basis of several interesting graph filters. The main one is *dot*, an advanced directed graph layout program inspired by *dag* [GNV,EN]. Overall, using *libgraph* has been advantageous. Its performance is sufficient for creating production software, and we have been successful in creating graph filters to work with *dot*, as next described. This was never really possible with *dag*.

tred and *scc* are *dot* pre-processors that help users make more readable layouts of large graphs. Since graphs that arise from software engineering applications can be large, we are interested in practical techniques for partitioning, selecting, or collapsing graphs to cut them down to manageable size. As already mentioned, *tred* removes edges if their endpoints can be reached by another path in the graph. This was implemented by coding a standard algorithm in 50 lines of C. *scc* finds strongly connected components and makes them subgraphs. These subgraphs might then be collapsed or drawn as "clusters".

neato embeds undirected graphs using virtual physical models [KK]. By using *libgraph* interfaces, *neato* shares files with other tools, as well as code generators from *dot* that handle shapes, fonts, colors, and pagination in several graphics languages.

Finally, the interactive graph editor *dotty*, though not compiled with *libgraph*, uses its file format and so is compatible at the file and process level. *dotty* is written on top of *lefty*, a multiple view programmable graphics editor with an interpretive high-level procedural language [DK]. The graph viewer is implemented as a set of *lefty* scripts that define all aspects of graph presentation and interaction. Some C code was added to print and parse graph files using *lefty*'s built-in data structures (associative arrays).

Table 4 gives the timings of several benchmark programs on a few sample graphs. The number of nodes and edges in each graph is listed in parentheses. *dynamics* is a graph from the book

World Dynamics by J.W. Forrester. *fsm* is a control program graph for a digital signal processor. *vi* is the call graph of a well-known editor. *usa* is an old UUCP backbone map. All graphs except *dynamics* have 5 node attributes and 3 edge attributes.

The compiled size of all test programs on a Sun-4 was 72K. (libdict.a and libgraph.a are 10K and 75K, respectively.) *read* and *read and write* exercise the graph parser and printer. *strong components* reads a graph and writes it with the strong components made into subgraphs. It uses a linear-time algorithm [Sed]. "Transitive reduction" is very compute-bound, using an $O(|V|^3)$ algorithm (this is asymptotically optimal).

A number of other graph packages are worth mentioning in comparison. The C++ Graph library [We] is similar to *libgraph* in some ways. It stores nodes and edges in sets using hash tables. Though subgraphs are not part of its model, nodes and edges are stored as references and can be stored in multiple graphs. The main difficulties in using the C++ graph library are C++ itself (such as the macros or templates that create parameterized graph classes) and the absence of a graph I/O capability. A less important point is that we have found ordered sets to be useful in layout programs, where it is desirable to retain the input order of nodes and edges to control the layout. Other well-known graph programming packages, such as GraphEd [Him] and Edge [New] are much larger than *libgraph*, depend on specific versions of both C++ and X windows, and do not address graph I/O. Although substantial efforts, they do not offer the capabilities we need.

In our applications, we find *libgraph*'s is a good replacement for ad-hoc application-specific graph routines, in both performance and selection of features. One problem we did encounter is that in expensive loops, we may want to avoid the cost of a function call to visit each node or edge imposed by the *dnext()* function of *libdict*. For example, the graph drawing program *dot* has a costly inner loop that iteratively reduces edge crossings. To bypass *dnext()*, we created arrays of edge pointers in `nodeinfo`. These can be scanned quickly, but we have found that we miss the generality of *libdict* edge sets, and the work-around has caused more than its share of bugs. A more recent version of *libdict* does provide macro functions such as *dflatten()*, *dlink()*, and *dobj()* to traverse objects using the internal link list pointers. This will improve performance of `nextedge`, `nextnode`, etc.

Conclusions

We have described libraries for programming with dictionaries and graphs. The dictionary library provides a consistent interface for dealing with hashed and ordered dictionaries, and has an efficient implementation using self-adjusting data structures that adapt well to biased search patterns. The graph library offers good abstractions, and encourages the creation of compatible graph-processing programs.

For information on obtaining *libdict* or *libgraph*, please contact the authors.

References

[DK] Dobkin, D. and E. Koutsofios. LEFTY: A Two-view Editor for Technical Pictures, *Graphics Interface '91*, pp. 68–76, 1991.

[GNV] Gansner, E.R., S. C. North, and K.-P. Vo. DAG— A Program that Draws Directed Graphs, *Software Practice and Experience* 18(11), pp. 1047–1062, Nov 1988.

[Him] Himsolt, Michael. GraphEd 3.0, available by anonymous ftp to forwiss.uni-passau.de (132.231.1.10) in /pub/local/graphed.

[KK] Kamada, T., and S. Kawai. An algorithm for drawing general undirected graphs, *Information Processing Letters* 31(1), pp. 7–15, April 1989.

[Koe] Koenig, A.R. Associative Arrays in C++, Proceedings of Summer 1988 USENIX Conference.

[KN] Koutsofios, E. and S. C. North. Drawing graphs with dot, unpublished Technical Report available from the authors, ATT Bell Laboratories, Murray Hill, N.J., 1992.

[Knu] Knuth, D.E. *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, 1973.

[KV] Korn, D.G. and K.-P. Vo. SFIO: Safe/Fast String/File IO, Proceedings of Summer 1991 Usenix Conference.

[New] Newbery, F. An Interface Description Language for Graph Editors, *Proceedings of the Conference on Visual Programming Languages*, Pittsburg, PA, pp. 144–149, 1988.

[NNGS] Nestor, J. R., J.M. Newcomer, P. Giannini, and D.L. Stone. *IDL: The Language and Its Implementation*, Prentice Hall, 1990.

Program	dynamics(48,69)	fsm(159,205)	vi(457,1757)	usa(5940,14875)
read	0.19	0.55	2.27	14.83
read and write	0.23	0.71	2.90	19.26
strong components	0.27	0.74	3.23	26.07
transitive reduction	0.33	3.16	761.07	(NA)

Table 4: Libgraph Benchmarks (Elapsed Real Time)

[Sed] Sedgewick, Robert. *Algorithms*, 2nd Ed. Addison-Wesley, 1988.

[SV] Unix System V Release 4 Programmer's Reference Manual, Prentice Hall, 1990.

[ST] Sleator, D.D. and R.E. Tarjan. Self-adjusting binary search trees, *JACM* 32(3) pp. 652–686, 1985.

[We] Weitzen, Terry. The C++ Graph Classes: A Tutorial, in *C++ Standard Components Programmer's Guide*, Unix System Laboratories, 1992.

Author Information

Stephen North received an M.A. and a Ph.D. in Computer Science from Princeton University in 1983 and 1986. He has been a Member of Technical Staff at AT&T Bell Laboratories at Murray Hill since 1980. His research interests include graph layout programs and algorithms and interactive programming environments. His electronic mail address is north@ulysses.att.com.

Phong Vo received an M.A. and a Ph.D. in Mathematics from the University of California at San Diego in 1977 and 1981. He has been with AT&T Bell Laboratories at Murray Hill since 1981 and is currently a Distinguished Member of Technical Staff. His research interests include graph theory, data structures and algorithms, user interface, and software tools. He authored or coauthored a number of popular UNIX tools including the latest System V malloc memory allocation package, <curses> library for screen management and IFS, a language for building applications with menu and form interfaces. He was named an AT&T Bell Laboratories Fellow in 1992. Reach him electronically at kpv@ulysses.att.com.

Linking Shared Segments

W. E. Garrett, M. L. Scott, R. Bianchini, L. I. Kontothanassis, R. A. McCallum, J. A. Thomas, R. Wisniewski, & S. Luk – University of Rochester

ABSTRACT

As an alternative to communication via messages or files, shared memory has the potential to be simpler, faster, and less wasteful of space. Unfortunately, the mechanisms available for sharing in Unix are not very easy to use. As a result, shared memory tends to appear primarily in self-contained parallel applications, where library or compiler support can take care of the messy details. We have developed a system, called Hemlock, for transparent sharing of variables and/or subroutines across application boundaries. Our system is backward compatible with existing versions of Unix. It employs dynamic linking in conjunction with the Unix *mmap* facility and a kernel-maintained correspondence between virtual addresses and files. It introduces the notion of *scoped linking* to avoid naming conflicts in the face of extensive sharing.

1. Introduction

Multi-user operating systems rely heavily on the ability of processes to interact with one another, both within multi-process applications and between applications and servers of various kinds. In the Unix world, processes typically interact either through the file system, or via some form of message passing. Both mechanisms have their limitations, however, and support for a third approach — shared memory — can also be extremely useful.

Memory sharing between arbitrary processes is at least as old as Multics[17]. It suffered something of a hiatus in the 1970s, but has now been incorporated into most variants of Unix. The Berkeley *mmap* facility was designed, though never actually included, as part of the 4.2BSD and 4.3BSD releases[12]; it appears in several commercial systems, including SunOS. ATT's *shm* facility became available in Unix System V and its derivatives. More recently, memory sharing via inheritance has been incorporated in the versions of Unix for several commercial multiprocessors, and the external pager mechanisms of Mach[1] and Chorus[18] can be used to establish data sharing between arbitrary processes.

Shared memory has several important advantages over interaction via files or messages.

1. Many programmers find shared memory more conceptually appealing than message passing. The growing popularity of distributed shared memory systems[16] suggests that programmers will adopt a sharing model even at the expense of performance.
2. Shared memory facilitates transparent, asynchronous interaction between processes, and shares with files the advantage of not requiring that the interacting processes be active concurrently.
3. When interacting processes agree on data formats and virtual addresses, shared memory provides a means of transferring information from one process to another without translating it to and from a (linear) intermediate form. The code required to save and restore information in files and message buffers is a major contributor to software complexity, and much research has been aimed at reducing this burden (e.g., through data description languages and RPC stub generators).
4. When supported by hardware, shared memory is generally faster than either messages or files, since operating system overhead and copying costs can often be avoided. Work by Bershad and Anderson, for example[4], indicates that message passing should be built on top of shared memory when possible.
5. As an implementation technique, sharing of read-only objects can save significant amounts of disk space and memory. All modern versions of Unix arrange for processes executing the same load image to share the physical page frames behind their text segments. Many (e.g., SunOS and SVR4) extend this sharing to dynamically-linked position-independent libraries. More widespread use of position-independent code, or of logically-shared, re-entrant code, could yield additional savings.

Both files and message passing have applications for which they are highly appropriate. Files are ideal for data that have little internal structure, or that are frequently modified with a text editor. Messages are ideal for RPC and certain other common patterns of process interaction. At the same time, we believe that many interactions currently achieved through files or message passing could better be expressed as operations on shared data. Many of the files described in section 5 of the Unix manual, for example, are really long-lived data structures. It seems highly inefficient, both computationally and in

terms of programmer effort, to employ access routines for each of these objects whose sole purpose is to translate what are logically shared data structure operations into file system reads and writes. In a similar vein, we see numerous opportunities for servers to communicate with clients through shared data rather than messages, with savings again in both cycles and programmer effort.

Despite its merits, however, shared memory in Unix remains largely confined to inheritance-based sharing within self-contained multiprocessor applications, and special-purpose sharing with devices. We speculate that much of the reason for this limited use lies in the lack of a transparent interface: access to private memory is much simpler and more easily expressed than access to shared memory; sharing is difficult to set up in the first place and variables and functions in shared memory cannot be named directly.

Both the System V *shm* and Berkeley *mmap* facilities require the user to know significant amounts of set-up information before sharing can take place. Processes must agree on ownership of a shared segment, and (if pointers are to be used) on its location in their respective address spaces. Processes using *shm* must also agree on some form of naming convention to identify shared segments (*mmap* uses file system naming). Most important, neither *mmap* nor *shm* allows language level access to shared segments. References to shared variables and functions must in most languages (including C) be made indirectly through a pointer. There is no performance cost for this indirection on many machines, but there is a loss in both transparency and type safety—static names are not available, explicit initialization is required, and any substructure for the shared memory is imposed by convention only.

In an attempt to address these problems we have developed a system, Hemlock,[1] that automates the creation and use of shared segments. Our goal in developing Hemlock was to simplify the interface to shared memory facilities while increasing the flexibility of the shared memory segments at the same time. Hemlock consists of new static and dynamic linkers, a run-time library, and a set of kernel extensions. These components cooperate to map and link shared segments into programs, providing type safety and language-level access to shared objects, and hiding the distinction between shared and private objects. Hemlock also facilitates the use of pointers to shared objects by maintaining a special file system, with a globally-consistent mapping between virtual addresses and sharable files. The mapping ensures that a given shared object lies at the same virtual address in every address space. Finally, through its *lazy dynamic linking*, Hemlock allows the programmer to design applications whose components, both private and shared, are determined at run time.

We focus in this paper on linker support for sharing, including *scoped linking* to avoid the naming conflicts that arise when linking across conventional application boundaries, *dynamic linking* to permit the private and shared components of applications to be determined at run time, and *lazy linking* to minimize unnecessary work. We provide an overview of Hemlock in section 2, and a more detailed description of its linkers in section 3. We describe example applications in section 4, discuss some semantic subtleties in section 5, and conclude in section 6.

2. An Overview of Hemlock

Our emphasis on shared memory has its roots in the Psyche project[19, 20]. Our focus in Psyche was on mechanisms and conventions that allow processes from dissimilar programming models (e.g., Lynx threads and Multilisp futures) to share data abstractions, and to synchronize correctly[14, 21]. The fundamental assumption of this work was that sharing would occur both within and among applications. Our current work[7, 23] is an attempt to make that sharing commonplace in the context of traditional operating systems.

Hemlock uses dynamic linking to allow processes to access shared code and data with the same syntax employed for private code and data. It also places shared segments into a special file system that maintains a globally-consistent mapping between sharable objects and virtual addresses, thereby ensuring that pointers to shared objects will be interpreted consistently in different protection domains. Unlike the "shared" libraries of systems such as SunOS and SVR4, Hemlock supports genuine write sharing, not just the physical sharing of logically private pages. Unlike such integrated programming environments as Cedar[24] and Emerald[10], it supports sharing of modules written in conventional languages, in a manner that is backward compatible with Unix. An early prototype of Hemlock ran under SunOS, but we are now working on Silicon Graphics machines (with SGI's IRIX operating system). Our long-term plans call for the exploitation of processors with 64-bit addressing, but this is beyond the scope of the current paper.

We use the term *segment* to refer to what Unix and Mach call a "memory object". Each segment can be accessed as a *file* (with the traditional Unix interface), or it can be mapped into a process's address space and accessed with load and store instructions. A segment that is linked into an address space by our static or dynamic linkers is

[1] Named for an evergreen tree species common in upstate New York, and for one of the lakes from which Rochester obtains its water supply.

referred to as a *module*. Each module is created from a *template* in the form of a Unix .o file. Each template contains references to *symbols*, which are names for *objects*, the items of interest to programmers. (Objects have no meaning to the kernel.) The linkers cooperate with the kernel to assign a virtual address to each module. They *relocate* modules to reside at particular addresses (by finalizing absolute references to internal symbols; some systems call this *loading*), and they *link* modules together by resolving cross-module references.

Our linkers associate a shared segment with a Unix .o file, making it appear to the programmer as if that file had been incorporated into the program via separate compilation (see Figure 1). Objects (variables and functions) to be shared are generally declared in a separate .h file, and defined in a separate .c file (or in corresponding files of the programmer's language of choice). They appear to the rest of the program as ordinary external objects. The only thing the programmer needs to worry about (aside from algorithmic concerns such as synchronization) is a few additional arguments to the linker; no library or system calls for set-up or shared-memory access appear in the program source.

Hemlock's linker for sharing, *lds*, is currently implemented as a wrapper that extends the functionality of the Unix *ld* linker. Lds defines four *sharing classes* for the object modules (.o files) from which an executing program is constructed. These classes are *static private*, *dynamic private*, *static public*, and *dynamic public*. Classes can be specified on a module-by-module basis in the arguments to lds. They differ with respect to the times at which they are created and linked, and the way in which they are named and addressed; see Table 1[2].

At static link time, lds creates a load image containing a new instance of every private static module. It also creates any public static modules that do not yet exist, but leaves them in separate files; it does not copy them into the load image. A public module resides in the same directory as its template (.o) file, and has a name obtained by dropping the final '.o'. It also has a unique, globally-

[2]For the purposes of this paper, we use the word 'process' in the traditional Unix sense. Like most researchers, we believe that operating systems should provide separate abstractions for threads of control and protection domains. Our work is compatible with this separation, but does not depend upon it.

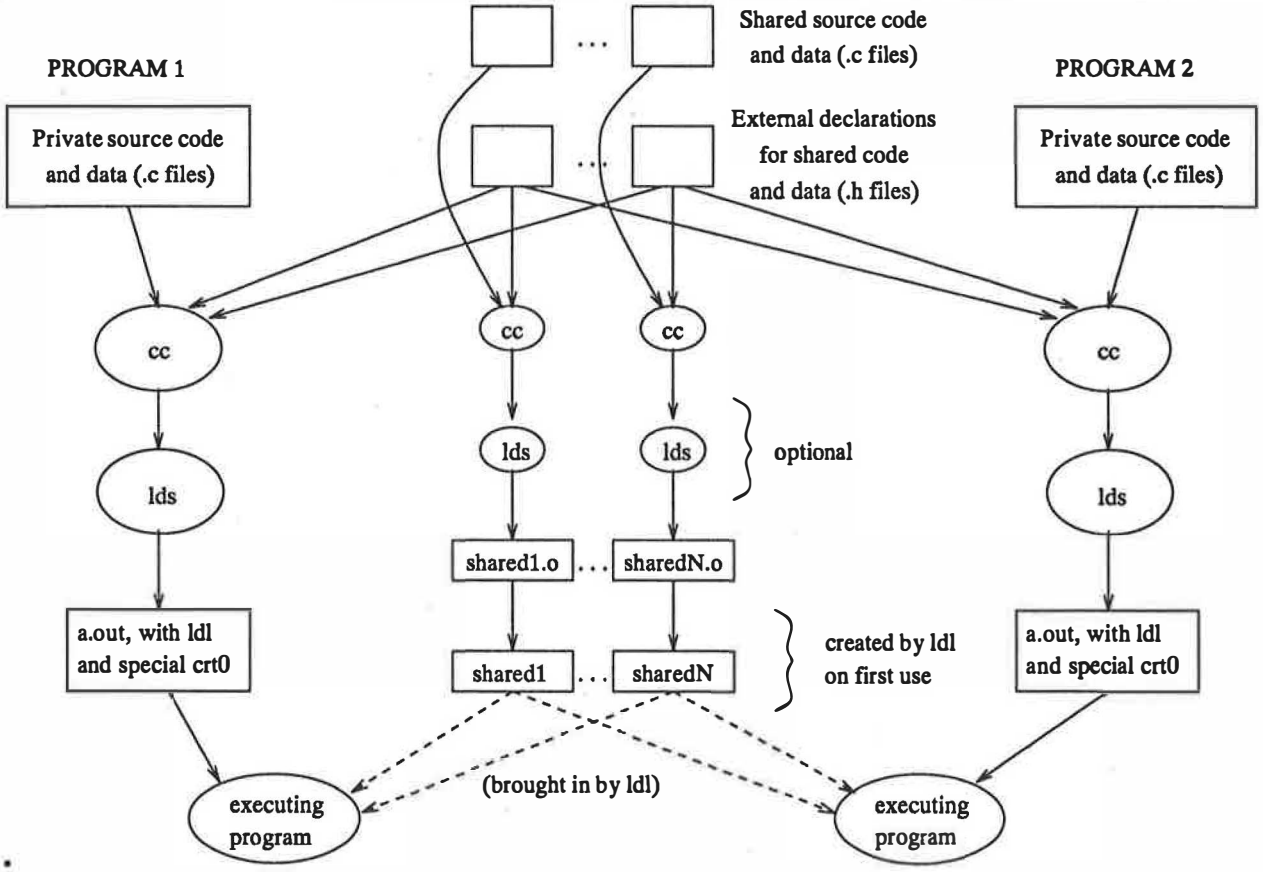


Figure 1: Building a Program with Linked-in Shared Objects

agreed-upon virtual address, and is internally relocated on the assumption that it resides at that address. Public modules are persistent; like traditional files they continue to exist until explicitly destroyed.

Lds resolves undefined references to symbols in static modules. It does not resolve references to symbols in dynamic modules. In fact, it does not even attempt to determine which symbols are in which dynamic module, or insist that the modules yet exist. Instead, lds saves the module names and search path information in the program load image, and links in an alternative version of crt0.o, the Unix program start-up module. At run time, crt0 calls our lazy dynamic linker, *ldl*.

Ldl uses the saved information to locate dynamic modules. It creates a new instance of each dynamic private module, and of each dynamic public module that does not yet exist. It then maps static public modules and all dynamic modules into the process address space, and resolves undefined references from the main load image to objects in the dynamic modules. If any module contains undefined references (this is likely for dynamic private modules, and possible for newly-created public modules), ldl maps the module without access permissions, so that the first reference will cause a segmentation fault. It installs a signal handler for this fault. When a fault occurs, the signal handler resolves any undefined external references in (all pages of) the module that has just been accessed, mapping in (possibly inaccessibly) any new modules that are needed.

This *lazy linking* supports a programming style in which users refer to modules, symbolically, throughout their programming environment. It allows us to run processes with a huge "reachability graph" of external references, while linking only the portions of that graph that are actually used during any particular run. We envision, for example, rewriting the *emacs* editor with a functional interface to which every process with a text window can be linked. With lazy linking, we would not bother to bring the editor's more esoteric features into a particular process's address space unless and until they were needed.

At static link time, modules are specified to lds the same way they are specified to ld: as absolute or relative path names. When attempting to find modules with relative names, lds uses a search path that can be altered by the user. It looks first in the current directory, next in an optional series of directories specified via command-line arguments, then in an optional series of directories specified via an environment variable, and finally in a series of default directories. Lds applies the search strategy at static link time for modules with a static sharing class. It passes a description of the search strategy to ldl for use in finding modules with a dynamic sharing class.

A template (.o) file is generally produced by a compiler. In addition, it can at the user's discretion be run through lds, with an argument that retains relocation information. In this case, lds can be asked to include search strategy information in the new .o file. When creating a new dynamic module from its template at run time, ldl attempts to resolve undefined references out of the new module using the search strategy (if any) specified to lds when creating that module. If this strategy fails, it reverts to the strategy of the module(s) that make references into the new module. This *scoped linking* preserves abstraction by allowing a process to link in a large subsystem (with its own search rules), without worrying that symbols in that subsystem will cause naming conflicts with symbols in other parts of the program. Scoped linking is discussed in further detail in the following section.

To facilitate the use of pointers, we must insist that all public modules be linked at the same virtual address in every protection domain. To ensure such *uniform addressing* on a 64-bit machine, we would associate a unique range of virtual addresses with every Unix file. On 32-bit machine, we maintain addresses only for files on a special disk partition, and then insist that public modules (and the templates from which they are created) reside on this partition. We retain the traditional Unix interfaces to the shared file system, both for the sake of backward compatibility and because we believe that these interfaces are appropriate for many applications.

The user-level handler for the SIGSEGV signal catches references to modules that are not currently

Sharing Class	When linked	New instance created/destroyed for each process	Default portion of address space
Static private Dynamic private	Static link time Run time	yes	Private
Static public Dynamic public	Static link time Run time	no	Public

Table 1: Class creation and link times

part of the address space of the executing process. The handler actually serves two purposes: it cooperates with ldl to implement lazy linking, and it allows the process to follow pointers into segments that may or may not yet be mapped. When triggered, the handler checks to see if the faulting address lies in the shared portion of the process's address space. If so, it uses a (new) kernel call to translate the address into a path name and, access rights pemitting, maps the named segment into the process's address space. If the address lies in a module that has been set up for lazy linking, the handler invokes ldl to resolve any undefined or relocatable references. (These may in turn cause other modules to be set up for lazy linking.) Otherwise, the handler opens and maps the file. It then restarts the faulting instruction. For compatibility with programs that already catch the SIGSEGV signal, the library containing our signal handler provides a new version of the standard *signal* library call. When the dynamic linking system's fault handler is unable to resolve a fault, a program-provided handler for SIGSEGV is invoked, if one exists.

3. Linking in Hemlock

Linker support for sharing capitalizes on the lowest common denominator for language implementations: the object file. By making modules correspond to object files, Hemlock gives the programmer first-class access to the objects they contain — with language-level naming, type checking, and scope rules — without modifying the compilers. By comparison, sharing based on pointer-returning system calls is comparatively distant from the programming language. The subsections below provide additional detail on the linkers, the shared file system, and the rationale for lazy and scoped linking.

The Linkers

Our current static linker is implemented as a wrapper, lds, around the standard IRIX ld linker. The wrapper processes new command line options directly related to its functionality and passes the others to ld. Lds-specific options allow for the association of sharing classes with modules and the specification of search paths to be used when locating modules. In addition, lds provides ldl with relocation information about static modules and warns the user if the dynamic modules do not yet exist. We are in the process of building a completely new stand-alone static linker that will also support scoped linking, currently available only in the dynamic linker, ldl.

Both lds and ldl use an extended search strategy for modules, inspired by the analogous strategy in the SunOS dynamic linker. At static link time, lds searches for modules in (1) the current directory, (2) the path specified in a special command-line argument, (3) the path specified by the LD_LIBRARY_PATH environment variable, and (4) the default library directories. If there is more than one static module with the same name, lds uses the first one it finds. At execution time, ldl searches for dynamic modules in (1) the path specified by the LD_LIBRARY_PATH environment variable, and (2) the directories in which lds searched for static modules: the directory in which static linking occurred, the directories specified on the lds command line, the directories specified by the LD_LIBRARY_PATH variable at static link time, and the default directories. Users can arrange to use new versions of dynamic modules by changing the LD_LIBRARY_PATH environment variable prior to execution. This feature is useful for debugging and, more important, for customizing the use of shared data to the current user or program instance. (We return to this issue in section 4 below.) Lds aborts linking if it cannot find a given static module. It issues a warning message and continues linking if it cannot find a given dynamic module.

To support the dynamic linker, lds creates a data structure listing the dynamic modules, and describing the search path it used for static modules. To give ldl a chance to run prior to normal execution, lds links C programs with a special start-up file. It would use similar files for other programming languages. Ldl also creates any static public modules that do not yet exist, and initializes those objects from their templates. Finally, in the current wrapper-based implementation, lds must compensate for certain shortcomings of the IRIX ld. Ld refuses to retain relocation information for an executable program, so lds must save this in an explicit data structure. Ld also refuses to resolve references to symbols at absolute addresses (as required for static public modules), so lds must do so.

Ldl differs from most dynamic linkers in several ways. Its facilities for lazy and scoped linking are discussed in more detail below. In addition, it will use symbols found in dynamically-linked modules to resolve undefined references in the statically-linked portion of the program, even when the location of those symbols was not known at static link time. To cope with an unfortunate limitation of the R3000 architecture, ldl insists that modules be compiled with a flag that disables use of the processor's performance-enhancing global pointer register. Addressing modes that use the pointer are limited to 24 bit offsets, and are incompatible with a large sparse address space. To cope with a similar 28-bit addressing limit on the processor's jump instructions, lds and ldl arrange for over-long branches to be replaced with jumps to new, nearby code fragments that load the appropriate target address into a register and jump indirectly.

Address Space and File System Organization

Given appropriate rights, programs should be able to access a shared object or segment simply by using its name. But different kinds of names are useful for different purposes. For human beings, ease of use generally implies symbolic names, both for objects and for segments: the linkers therefore accept file system names for segments, and support symbolic names for objects. For running programs, on the other hand, ease of use generally implies addresses: programs need to be able to follow pointers, even if they cross segment boundaries. It is easy to envision applications in which both types of names are useful. Any program that shares data structures and also manipulates segments as a whole may need both sets of names.

In our 32-bit prototype, we have reserved a 1G-byte region between the Unix heap and stack segments, and have associated this region with the kernel-maintained shared file system. The file system is configured to have exactly 1024 inodes, and each file is limited to a maximum of 1M bytes in size. Hard links (other than '.' and '..') are prohibited, so there is a one-one mapping between inodes and path names. We have modified the IRIX kernel to keep track of the mapping internally, and have provided system calls that translate back and forth.

All of the normal Unix file operations work in the shared file system. The only thing that sets it apart is the association between file names and addresses. Mapping from file names to addresses is easy: the *stat* system call already returns an inode number. We provide a new system call that returns the filename for a given inode, and we overload the arguments to *open* so that the programmer can open a file by address instead of by name, with a single system call. For the sake of simplicity, the mapping in the kernel from addresses to files employs a linear lookup table. We initialize the table at boot time by scanning the entire shared file system, and update it as appropriate when files are created and destroyed. For an experimental prototype, these measures have the desirable property of allowing the filename/address mapping to survive system crashes without requiring modifications to on-disk data structures or to utilities like *fsck* that understand those structures.

With 64-bit addresses, we will extend the shared file system to include all of secondary store, and will relax the limits on the number and sizes of shared files. We plan to provide every segment, whether shared or not, with a unique, system-wide virtual address. At the same time, we plan to retain the ability to overload addresses within a reserved, private portion of the 64-bit space. Within the kernel, we will abandon the linear lookup table and the direct association between inode numbers and addresses. Instead, we will add an address field to the on-disk version of each inode, and will link these inodes into a lookup structure — most likely a B-tree — whose presence on the disk allows it to survive across re-boots.

Lazy Dynamic Linking

Public modules in Hemlock can be linked both statically and dynamically. The advantage of dynamic linking is that it allows the makeup of a program to be determined very late. With dynamic linking, an application can be composed of different modules from run to run, depending on who is running it, what directories and modules currently exist, what changes have recently been made to environment variables, etc.

We expect to rely on run-time identification of modules for a variety of purposes. By using search paths containing directories that are named relative to the current or home directory, we can arrange for applications to link in data structures that are shared with other applications belonging to the same user, project etc. Similarly, by modifying environment variables prior to execution, we can arrange for new processes to find shared data in a temporary directory. We describe the use of this technique in parallel applications in section 4 below.

Dynamic linking is already used in several Unix systems (including SunOS and SVR4) to save space in the file system and in physical memory, and to permit updating of libraries without recompiling all the programs that employ them. In many of these systems, position-independent code (PIC) permits the text pages of libraries to be physically shared, but this is only an optimization; each process has a private copy of any static variables. The PIC produced by the Sun compilers uses jump tables that allow functions to be linked lazily, but references to data objects are all resolved at load time. Sun's ld also insists that all dynamically-linked libraries exist at static link time, in order to verify the names of their entry points.

Hemlock uses dynamic linking for both private and shared data, and does not insist on knowing at static link time which symbols will be found in which dynamically-linked modules. This latter point may delay the reporting of errors, and can increase the cost of run-time linking, but increases flexibility. Lds requires only that the user specify the names of all modules containing symbols accessed directly from the main load image. It then accepts arguments that allow the user to specify a search path on which to look for those modules at run time. Any module found may in turn specify a search path on which to look for modules containing symbols that *it* references.

Our fault-driven lazy linking mechanism is slower than the jump table mechanism of SunOS, but works for both functions and data objects, and does not require compiler support. We do not currently share the text of private modules, but will

do so when PIC-generating compilers become available under IRIX. Given the opportunity, we will adopt the SunOS jump-table-based lazy linking mechanism as an optimization: modules first accessed by calling a (named) function will be linked without fault-handling overhead.

Several dynamic linkers, including the Free Software Foundation's dld[9] and those of SunOS and SVR4, provide library routines that allow the user to link object modules into a running program. Dld will resolve undefined references in the modules it brings in, allowing them to point into the main program or into other dynamically-loaded modules. The Sun and SVR4 routines (*dlopen* and *dlsym*) do not provide this capability; they require the newly-loaded module be self-contained. Neither dld nor the explicitly-invoked Sun/SV routines resolves undefined references in the main program; they simply return pointers to the newly-available symbols.

Scoped Linking

Traditional linking systems, both static and dynamic, deal only with private symbols. They bind all external references to a given name to the same object in all linked modules. If more than one module exports an object with a given name, the linker either picks one (e.g., the first) and resolves all references to it, or reports an error. Our system of dynamic linking, with shared symbols and recursive, lazy inclusion of modules, presents cases where either behavior is undesirable.

Specifying that a module is to be included in a program starts a link in a potentially long chain. Hemlock allows modules to have their own search path and list of modules, which in turn may have their own lists, recursively. Linking a single module may therefore cause a chain reaction that ends up incorporating modules that the original programmer knew nothing about. These modules may have external symbols that the original program knew nothing about. Some of these external symbols may have the same name as external symbols exported by the main program, even though they are actually unrelated. This possibility introduces a potentially serious naming conflict.

The problem is that linkers map from a rich hierarchy of abstractions to a flat address space.

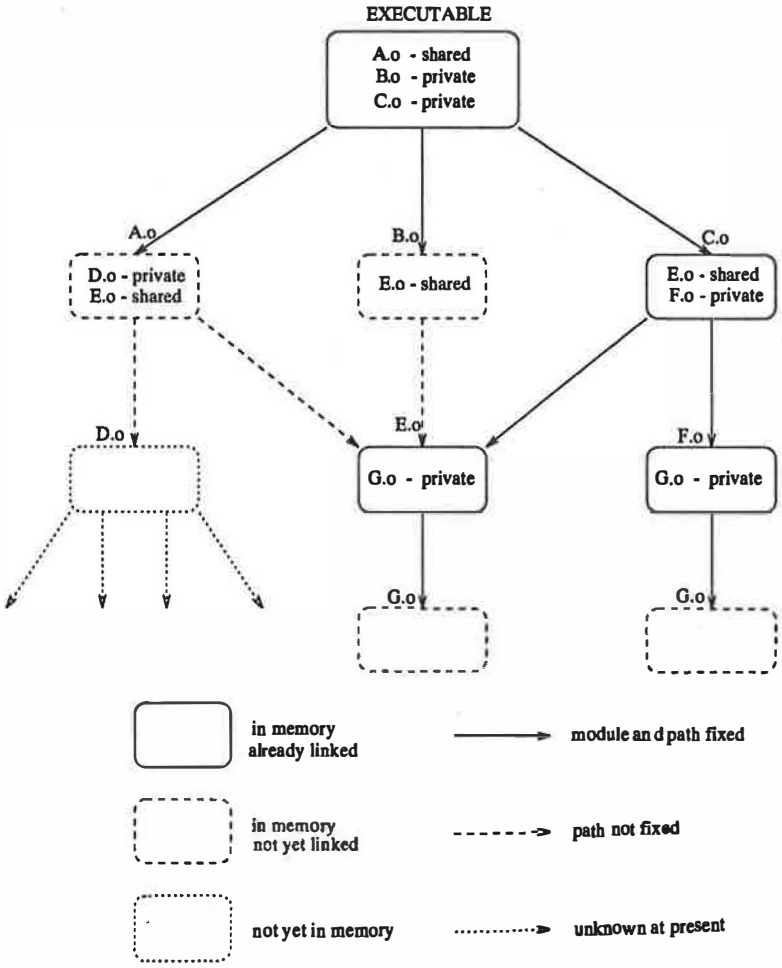


Figure 2: Hierarchical Inclusion of Dynamically-Linked Modules

Various programming languages (e.g., Modula-2 and Common Lisp) that use the idea of a module for abstraction already deal with this problem. Their implementations typically preface variables and function names with module names, thereby greatly reducing the chance of naming conflicts. Scoped linking provides similar freedom from ambiguity, in a language-independent way.

When a module M is brought in, its undefined references are first resolved against the external symbols of modules found on M's own module list and search path. If this step is not completely successful, consideration moves up to the module(s) that caused M to be loaded in—M's "parent", so to speak: remaining undefined references are resolved against the external symbols of modules found on the parent's module list and search path. If unresolved references still remain, they are then resolved using the module list and search path of M's grandparent, and so on.

The linking structure of a program can be viewed as a DAG (see Figure 2), in which children can search up from their current position to the root, but never down. Modules wishing to have control over their symbols must specify appropriate modules and directories on their module list and search path. Modules wishing to rely on a symbol being resolved by the parent can simply neglect to provide this information. References that remain undefined at the root of the DAG are left unresolved in the running program. If encountered during execution they result in segmentation faults that are caught by the signal handler, and could be used (at the programmer's discretion) to trigger application-specific recovery.

4. Example Applications

In this section we consider several examples of the usefulness of cross-application shared memory.

Administrative Files

Unix maintains a wealth of small administrative files. Examples include much of the contents of /etc, the score files under /usr/games, the many "dot" files in users' home directories, bitmaps, fonts, and so on. Most of these files have a rigid format that constitutes either a binary linearization or a parsable ASCII description of a special-purpose data structure. Most are accessed via utility routines that read and write these on-disk formats, converting them to and from the linked data structures that programs really use.

For the designer of a new structure, the avoidance of translation may not be overwhelming, but it is certainly attractive. As an example of the possible savings in complexity and cost, consider the *rwhod* daemon. Running on each machine, *rwhod* periodically broadcasts local status information (load average, current users, etc.) to other machines, and receives analogous information from its peers. As originally conceived, it maintains a collection of local files, one per remote machine, that contain the most recent information received from those machines. Every time it receives a message from a peer it rewrites the corresponding file. Utility programs read these files and generate terminal output. Standard utilities include *rwho* and *ruptime*, and many institutions have developed local variants. Using the early prototype of our tools under SunOS, we re-implemented *rwhod* to keep its database in shared memory, rather than in files, and modified the various lookup utilities to access this database directly. The result was both simpler and faster. On our local network of 65 *rwhod*-equipped machines, the new version of *rwho* saves a little over a second each time it is called. Though not earthshaking, this savings may be significant: many members of our department run a windowing variant of *rwho* every 60 seconds. We are currently porting the new server and utilities to our SGI-based system.

Utility Programs and Servers

Traditionally, UNIX has been a fertile environment for the creation and use of small tools that can be connected together, e.g., via pipes. Other systems, including Multics and the various open operating systems[24, 25] encourage the construction of similar building blocks at the level of functions, rather than program executables. In future work, we plan to use Hemlock facilities to experiment with functional building blocks in Unix. We also plan to experiment with the use of shared data to improve the performance of interfaces between servers and their clients.

When synchronous interaction is not required, modification of data that will be examined by another process at another time can be expected to consume significantly less time than kernel-supported message passing or remote procedure calls. Even when synchronous communication across protection domains is required, sharing between the client and server can speed the call. In their work on lightweight and user-level remote procedure calls, Bershad et al. argue that high-speed interfaces permit a much more modular style of system construction than has been the norm to date[4]. The growing interest in *microkernels*[28] suggests that this philosophy is catching on. In effect, the microkernel argument is that the proliferation of boundaries becomes acceptable when crossing these boundaries is cheap. We believe that it is even more likely to become acceptable when the boundaries are blurred by sharing, and processes can interact without necessarily crossing anything.

Parallel Applications

A parallel program can be thought of as a collection of sequential processes cooperating to accomplish the same task. Threads in a parallel application need to communicate with their peers for

synchronization and data exchange. On a shared memory multiprocessor this communication occurs via shared variables. In most parallel environments global variables are considered to be shared between the the threads of an application while local variables are private to a thread. In systems like Presto[3], however, both shared *and* private global variables are permitted. Presto was originally designed to run on a Sequent multiprocessor under the Dynix operating system. The Dynix compilers provide language extensions that allow the programmer to distinguish explicitly between shared and private variables. The SGI compilers, on the other hand, provide no such support.

When we set out to port Presto to IRIX in the fall of 1991, the lack of compiler-supported language extensions became a major problem. The solution we eventually adopted was to explicitly place shared variables in memory segments shared between the processes running the application. Placement had to be done by editing the assembly code, and was extremely tedious when attempted by hand. We created a post-processor to automate this procedure; it is 432 lines long (including 105 lines of *lex* source), and consumes roughly one quarter to one third of total compilation time. It also embeds some compiler dependencies; we were forced to re-write it when a new version of the C compiler was released.

We are currently modifying our Presto implementation to use our dynamic linking tools. Selective sharing can be specified with ease. Shared variables must still be grouped together in a separate file, but editing of the assembly code is no longer required. The parent process of the application, which exists solely for set-up purposes, and does none of the application's work, does not link the shared data file. Rather, it creates a temporary directory, puts a symbolic link to the shared data template into this directory, and then adds the name of the directory to the LD_LIBRARY_PATH environment variable. At static link time, the child processes of the parallel application specify that the shared data structures should be linked as a dynamic public module. When the parent starts the children, they all find the newly-created symlink in the temporary directory. The first one to call ldl creates and initializes the shared data from the template, and all of them link it in.[3] When the computation terminates the parent process performs the necessary cleanup, deleting the shared segment, template symlink, and temporary directory.

Programs with Non-Linear Data Structures

Even when data structures are not accessed concurrently by more than one process, they may be shared sequentially over time. Compiler symbol tables are a canonical example. In a multi-pass compiler, pointer-rich symbol table information is often linearized and saved to secondary store, only to be reconstructed in its original form by a subsequent pass. The complexity of this saving and restoring is a perennial complaint of compiler writers, and much research has been devoted to automating the process[15].[4] Similar work has occurred in the message-passing community[8].

With pointers permitted in files, and with a global consensus on the location of every segment, pointer-rich data structures can be left in their original form when saved across program executions. Segments thus saved are position-dependent, but for the compiler writer this is not a problem; the idea is simply to transfer the data between passes.

In a related case study, we have examined our compiler for the Lynx distributed programming language[22], designed around scanner and parser generators developed at the University of Wisconsin. The Wisconsin tools produce numeric tables which a pair of utility programs translate into initialized data structures for separately-developed scanner and parser drivers, written in Pascal. Since Pascal lacks initialized static variables, the initialization trick depends on a non-portable correspondence in data structure layouts between C and Pascal.

With Hemlock, the utility programs that read the numeric output of the scanner and parser generators would share a persistent module (the tables) with the Lynx compiler. The utility programs would initialize the tables; the compiler would link them in and use them. These changes would eliminate between 20 and 25% of code in the utility programs. They would also save a significant amount of time: the C version of the tables is over 5400 lines, and takes 18 seconds to compile on a Sparcstation 1.

An additional example can be found in the *xfig* graphical editor, which we have re-written under Hemlock. While editing, xfig maintains a set of linked lists that represent the objects comprising a figure. It originally translated these lists to and from a pointer-free ASCII representation when reading and writing files. As the same time, xfig must be able to copy the pointer-rich representation, to duplicate objects in a figure. The Hemlock version of xfig uses the pre-existing copy routines for files, at a savings of over 800 lines of code.

5. Discussion

Public vs. Private Code and Data

A representation of addressing in Hemlock appears in Figure 3. The public portion of the address space appears the same in every process,

[3]Ldl uses file locking to synchronize the creation of shared segments.

[4]Some of this research is devoted to issues of machine and language independence, but much of it is simply a matter of coping with pointers.

though which of its segments are actually accessible will vary from one protection domain to another. Addresses in the private portion of the address space are overloaded; they mean different things to different processes. Private modules (including the main module of every process) are linked into the private, overloaded portion of the address space, while public modules are linked at their globally-understood address.

Every program begins execution in the private portion of the address space. In our current 32-bit system, only one quarter of the address space is public, and traditional, unmodified Unix programs never use public addresses. In a 64-bit system, the vast majority of the address space would be public, and we would expect programmers to gradually adopt a style of programming in which public addresses are used most of the time. Backward compatibility is thus the principal motivation for providing private addresses. Some existing programs (generally not good ones) assume that they are linked at a particular address. Most existing programs are created by compilers that use absolute addressing modes to access static data, and assume that the data are private. Many create new processes via *fork*.

Chase, et al.,[5] observe that the Unix fork mechanism is based in a fundamental way on the use of static, private data at fixed addresses. Their Opal system, which adopts a strict, single global translation, dispenses with fork in favor of an RPC-based mechanism for animating a newly-created protection domain. We adopted a similar approach in Psyche; we agree that fork is an anachronism. It works fine in Hemlock, however, and we retain it by weight of precedent. The child process that results from a fork receives a copy of each segment in the private portion of the parent's address space, and shares the single copy of each segment in the public portion of the parent's address space. In all cases, the parent and child come out of the fork with identical program counters. If the parent's PC was at a private address, the parent and child come out in logically private but identical copies of the code. If the parent's PC was at a public address, the parent and child come out in logically shared code, which must be designed for concurrent execution in order to work correctly.

Like Psyche, Hemlock adopts the philosophy that code should be considered shared precisely when its static data is shared. Under this philosophy, the various implementations of "shared"

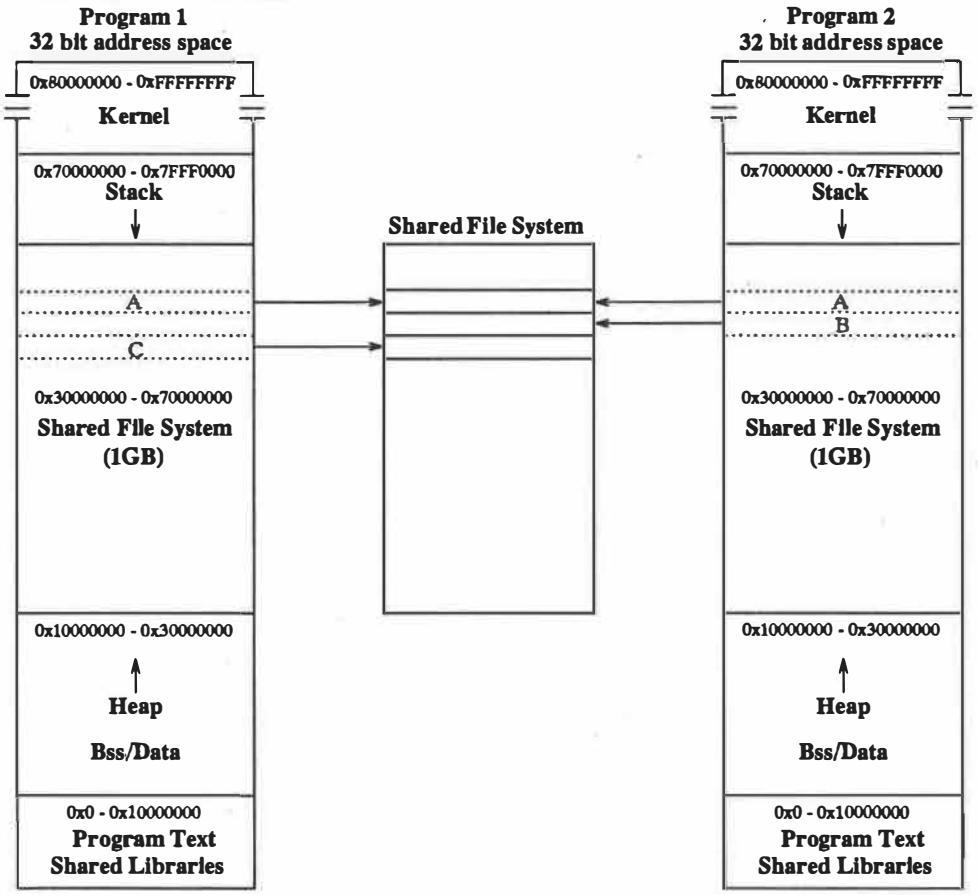


Figure 3: Hemlock Address Spaces

libraries in Unix are in fact space-saving implementations of logically *private* libraries. There is no philosophical difference between these implementations and the much older notion of "shared text"; one is implemented in the kernel and the other in the linkers, but both serve to conserve physical page frames while allowing the programmer to ignore the existence of other processes.

A different philosophical position is taken in systems such as Multics[17], Hydra[27], and Opal, which clearly separate code from data and speak explicitly of processes executing in shared code but using private (static) data. Multics employs an elaborate hardware/software mechanism in which references to static data are made indirectly through a base register and process-private link segment. Hydra employs a capability-based mechanism implemented by going through the kernel on cross-segment subroutine calls. Opal postulates compilers that generate code to support the equivalent of Multics base registers in an unsegmented 64-bit address space.

With most existing Unix compilers, processes executing the same code at the same address will access the same static data, unless the data addresses are overloaded. This behavior is consistent with the Hemlock philosophy. Code in the private portion of the address space is private; if it happens to lie at the same physical address as similar-looking code in another address space (as in the case of Unix shared text), the overloading of private addresses still allows it to access its own copy of the static data. Code in the public portion of the address space is shared if and only if more than one process chooses to execute it, in which case all processes access the same static data.

In practice, we can still share physical pages of code between instances of the same module by using position-independent code (PIC), which embeds no assumptions (even after linking) about the address at which it executes or about the addresses of its static data or external code or data. Compilers that generate linkage-table-based PIC are already used for shared libraries in SunOS and SVR4, and will soon be available under IRIX.[5]

The decision as to whether sharable code at a given virtual address always accesses the same static data is essentially a matter of taste; we have adopted a philosophy more in keeping with Unix than with Multics. In code that is logically shared (with static data that is shared), Hemlock programmers can differentiate between processes on the basis of

- parameters passed into the code in registers, or in an argument record accessed through a register (frame pointer),
- return values from system calls that behave differently for different processes (possible only if processes are managed by the kernel),
- explicit, programmer-specified overloading of (a limited number of) addresses, or
- programming environment facilities (e.g., environment variables) implemented in terms of one of the above.

[5]We should emphasize that our system does not *require* PIC. In fact, the SGI compilers don't produce it yet. When it becomes available we will obtain no new functionality, but we will use less space.

Caveats

Easy sharing is unfortunately not without cost. Although we firmly believe that increased use of cross-application shared memory can make Unix more convenient, efficient, and productive, we must also acknowledge that sharing places certain responsibilities on the programmer, and introduces problems.

Synchronization

Files are seldom write-shared, and message passing subsumes synchronization. When accessing shared memory, however, processes must synchronize explicitly. Unix already includes kernel-supported semaphores. For lighter-weight synchronization, blocking mechanisms can be implemented in user space by providing standard interfaces to thread schedulers[13], and several researchers have demonstrated that spin locks can be used successfully in user space as well, by preventing, avoiding, or recovering from preemption during critical sections[2, 6, 13], or by relinquishing the processor when a lock is unavailable[11].

Garbage Collection

When a Unix process finishes execution or terminates abnormally, its private segments can be reclaimed. The same cannot be said of segments shared between processes. Sharing introduces (or at least exacerbates) the problem of garbage collection. Good solutions require compiler support, and are inconsistent with the anarchistic philosophy of Unix. We see no alternative in the general case but to rely on manual cleanup. Fortunately, our shared file system provides a facility crucial for manual cleanup: the ability to peruse all of the segments in existence. Our hope is that the manual cleanup of general shared-memory segments will prove little harder than the manual cleanup of files, to which programmers are already accustomed.

Position-Dependent Files

As soon as we allow a segment to contain absolute internal pointers, we cannot change its address without changing its data as well. Files with internal pointers cannot be copied with *cp*, mailed over the Internet, or archived with *tar* and then restored in different places. Though many files need never move, in other cases the choice between being able to use pointers and being able to move and copy files may not be an easy one to make. Figures

from our modified version of xfig, for example, can safely be copied only by xfig itself.

Dynamic Storage Management

In the earlier overview section, we suggested that dynamic linking might encourage widespread re-use of functional interfaces to pre-existing utilities. It is likely that the interfaces to many useful functions will require variable-sized data structures. If the text editor is a function, for example, it will be much more useful if it is able to change the size of the text it is asked to edit. This suggests an interface based on, say, a linked list of dynamically-allocated lines, rather than a fixed array of bytes. We have developed a package designed to allocate space from the heaps associated with individual segments, instead of a heap associated with the calling program. This package is used by the Hemlock version of xfig. We expect that as we develop more applications we will be able to determine the extent and type of new storage management facilities that will be necessary.

Safety

It is possible that a programming error will cause a program to make an invalid reference to an address that happens to lie in a segment to which the user has access rights. Our signal handler will then erroneously map this segment into the running program and allow the invalid reference to proceed. We see no way to eliminate this possibility without severely curtailing the usefulness of our tools. The probability of trouble is small; the address space is sparse.

It is also possible that a program will circumvent our wrapper, execute a kernel call directly, and replace our signal handler. Since use of our tools is optional, we do not regard this as a problem; we assume that a program that uses our tools will use only the normal interface.

Finally, programming under Hemlock using shared memory requires a more defensive style of programming than is normally necessary when communicating via messages or RPC. It is easier to implement sanity checks for RPC parameters than it is to implement them for arbitrary shared data segments. Servers must be careful that their proper operation is not dependent on the proper operation of their clients.

Loss of Commonality

The ubiquity of byte streams and text files is a major strength of Unix. As shared-memory utilities proliferate, there is a danger that programmers will develop large numbers of incompatible data formats, and that the "standard Unix tools" will be able to operate on a smaller and smaller fraction of the typical user's data.

Many of the most useful tools in Unix are designed to work on text files. To the extent that persistent data structures are kept in a non-linear, non-text format, these tools become unusable. Administrative files, for example, are often edited by hand. There are good arguments for storing them as something other than ascii text, but doing so means abandoning the ability to make modifications with an ordinary text editor.

It is not entirely clear, of course, that most data structures *should* be modified with a text editor that knows nothing about their semantics. Unix provides a special locking editor (*vipw*) for use on /etc/passwd, together with a syntax checker (*ckpw*) to verify the validity of changes. System V employs a non-linear alternative to /etc/termcap (the *terminfo* database), and provides utility routines that translate to and from (with checking) equivalent ascii text.

Similar pros and cons apply to the design of programs as filters. The ability to pipe the output of one process into the input of another is a powerful structuring tool. Byte streams work in pipes precisely because they can be produced and consumed incrementally, and are naturally suited to flow control. Complex, non-linear data structures are unlikely to work as nicely. At the same time, a quick perusal of Unix directories confirms that many of the file formats currently in use have a rich, non-byte stream structure: *a.out* files, *ar* archives, *core* files, *tar* files, TeX *dvi* files, compressed files, inverted indices, the SunView defaults database, bitmap and image formats, and so forth.

6. Conclusion

Hemlock is a set of extensions to the Unix programming environment that facilitates sharing of memory segments across application boundaries. Hemlock uses dynamic linking to allow programs to access shared objects with the same syntax that they use for private objects. It includes a shared file system that allows processes to share pointer-based linked data structures without worrying that addresses will be interpreted differently in different protection domains. It increases the convenience and speed of shared data management, client/server interaction, parallel program construction, and long-term storage of pointer-rich data structures.

As of November 1992, we have a 32-bit version of Hemlock running on an SGI 4D/480 multiprocessor. This version consists of (1) extensions to the Unix static linker, to support shared segments; (2) a dynamic linker that finds and maps such segments (and any segments that they in turn require, recursively) on demand; (3) modifications to the file system, including kernel calls that map back and forth between addresses and path name/offset pairs in a dedicated shared file system, and (4) a fault handler that adds segments to a process's address space on demand, triggering the dynamic linker when appropriate.

Hemlock maintains backward compatibility with Unix, not only because we wish to retain the huge array of Unix tools, but also because we believe that the Unix interface is for the most part a good one, with a proven track record. We believe that backward compatibility has cost us very little, and has gained us a great deal. In particular, retention of the Unix file system interface, and use of the hierarchical file system name space for segments, provides valuable functionality. It allows us to use the traditional file read/write interface for segments when appropriate. It allows us to apply existing tools to segments. It provides a means of perusing the space of existing segments for manual garbage collection.

Problems that we are currently investigating include:

- Language Heterogeneity
  Hemlock uses the object file as a lowest common denominator among programming languages. It provides no magic, however, to ensure that object files produced by different compilers will embed compatible assumptions about the naming, types, and layout of shared data. These problems are not new of course; programs whose components are written in different languages, or compiled by different compilers, must already deal with the issue of compatibility. Problems are likely to arise more often, however, when sharing among multiple programs. We are interested in the possibility of automatically translating definitions of shared abstractions written in one language into definitions and optimized access routines written in another language.
- Synchronous Communication
  We plan to add a protection-domain switching system call to our modified IRIX kernel to support synchronous communication across protection boundaries in Hemlock. We speculate that the ability to migrate unprotected functionality into shared code will allow us in many cases to increase the degree of parallelism, and hence the performance, of fast RPC systems.
- Scoped Static Linking
  Because lds is implemented as a wrapper for ld, scoped linking is currently available in Hemlock only for dynamic modules. We plan to correct this deficiency in a new, fully-functional static linker.

Along with the above goals there are a number of other questions that we expect will be answered as we continue to build larger applications with Hemlock. These include:

- How important is the ability to overload virtual addresses? Is it purely a matter of backward compatibility?
- How best can our experience with Psyche (specifically, multi-model parallel programming and first-class user-level threads) be transferred to the Unix environment?
- To what extent can in-memory data structures supplant the use of files in traditional Unix utilities?
- In general, how much of the power and flexibility of open operating systems can be extended to an environment with multiple users and languages?

Many of the issues involved in this last question are under investigation at Xerox PARC (see [26] in particular). The multiple languages of Unix, and the reliance on kernel protection, pose serious obstacles to the construction of integrated programming environments. It is not clear whether all of these obstacles can be overcome, but there is certainly much room for improvement. We believe that shared memory is the key.

References

1. M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development," *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, pp. 93-112, June 1986.
2. T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism," *ACM Transactions on Computer Systems*, vol. 10, no. 1, pp. 53-79, February 1992. Originally presented at the *Thirteenth ACM Symposium on Operating Systems Principles*, 13-16 October 1991.
3. B. N. Bershad, E. D. Lazowska, H. M. Levy, and D. B. Wagner, "An Open Environment for Building Parallel Programming Systems," *Proceedings of the First ACM Conference on Parallel Programming: Experience with Applications, Languages and Systems*, pp. 1-9, New Haven, CT, 19-21 July 1988. In *ACM SIGPLAN Notices 23*:9.
4. B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy, "Lightweight Remote Procedure Call," *ACM Transactions on Computer Systems*, vol. 8, no. 1, pp. 37-55, February 1990. Originally presented at the *Twelfth ACM Symposium on Operating Systems Principles*, 3-6 December 1989.
5. J. S. Chase, H. M. Levy, M. Baker-Harvey, and E. D. Lazowska, "How to Use a 64-Bit Virtual Address Space," Technical Report 92-03-02, Department of Computer Science and Engineering, University of Washington, March 1992.

6. J. Edler, J. Lipkis, and E. Schonberg, "Process Management for Highly Parallel UNIX Systems," *Proceedings of the USENIX Workshop on Unix and Supercomputers*, Pittsburgh, PA, 26-27 September 1988. Also available as Ultracomputer Note #136, Courant Institute, N.Y.U., April 1988.
7. W. E. Garrett, R. Bianchini, L. Kontothanassis, R. A. McCallum, J. Thomas, R. Wisniewski, and M. L. Scott, "Dynamic Sharing and Backward Compatibility on 64-Bit Machines," TR 418, Computer Science Department, University of Rochester, April 1992.
8. M. Herlihy and B. Liskov, "A Value Transmission Method for Abstract Data Types," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 4, pp. 527-551, October 1982.
9. W. W. Ho and R. A. Olsson, "An Approach to Genuine Dynamic Linking," *Software—Practice and Experience*, vol. 21, no. 4, pp. 375-390, April 1991.
10. E. Jul, H. Levy, N. Hutchinson, and A. Black, "Fine-Grained Mobility in the Emerald System," *ACM Transactions on Computer Systems*, vol. 6, no. 1, pp. 109-133, February 1988. Originally presented at the *Eleventh ACM Symposium on Operating Systems Principles*, Austin, TX, 8-11 November 1987.
11. A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki, "Empirical Studies of Competitive Spinning for a Shared-Memory Multiprocessor," *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pp. 41-55, Pacific Grove, CA, 13-16 October 1991. In *ACM SIGOPS Operating Systems Review 25*:5.
12. S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, The Addison-Wesley Publishing Company, Reading, MA, 1989.
13. B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos, "First-Class User-Level Threads," *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pp. 110-121, Pacific Grove, CA, 14-16 October 1991. In *ACM SIGOPS Operating Systems Review 25*:5.
14. B. D. Marsh, C. M. Brown, T. J. LeBlanc, M. L. Scott, T. G. Becker, P. Das, J. Karlsson, and C. A. Quiroz, "Operating System Support for Animate Vision," *Journal of Parallel and Distributed Computing*, vol. 15, no. 2, pp. 103-117, June 1992.
15. C. R. Morgan, "Special Issue on the Interface Description Language IDL," *ACM SIGPLAN Notices*, vol. 22, no. 11, November 1987.
16. B. Nitzberg and V. Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms," *Computer*, vol. 24, no. 8, pp. 52-60, August 1991.
17. E. I. Organick, *The Multics System: An Examination of Its Structure*, MIT Press, Cambridge, MA, 1972.
18. M. Rozier and others, "Chorus Distributed Operating Systems," *Computing Systems*, vol. 1, no. 4, pp. 305-370, Fall 1988.
19. M. L. Scott, T. J. LeBlanc, and B. D. Marsh, "Design Rationale for Psyche, a General-Purpose Multiprocessor Operating System," *Proceedings of the 1988 International Conference on Parallel Processing*, vol. II – Software, pp. 255-262, St. Charles, IL, 15-19 August 1988.
20. M. L. Scott, T. J. LeBlanc, and B. D. Marsh, "Evolution of an Operating System for Large-Scale Shared-Memory Multiprocessors," TR 309, Computer Science Department, University of Rochester, March 1989.
21. M. L. Scott, T. J. LeBlanc, and B. D. Marsh, "Multi-Model Parallel Programming in Psyche," *Proceedings of the Second ACM Symposium on Principles and Practice of Parallel Programming*, pp. 70-78, Seattle, WA, 14-16 March, 1990. In *ACM SIGPLAN Notices 25*:3.
22. M. L. Scott, "The Lynx Distributed Programming Language: Motivation, Design, and Experience," *Computer Languages*, vol. 16, no. 3/4, pp. 209-233, 1991. Earlier version published as TR 308, "An Overview of Lynx," Computer Science Department, University of Rochester, August 1989.
23. M. L. Scott and W. Garrett, "Shared Memory Ought to be Commonplace," *Proceedings of the Third Workshop on Workstation Operating Systems*, Key Biscayne, FL, 23-24 April 1992.
24. D. Swinehart, P. Zellweger, R. Beach, and R. Hagmann, "A Structural View of the Cedar Programming Environment," *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 4, pp. 419-490, October 1986.
25. J. H. Walker, D. A. Moon, D. L. Weinreb, and M. McMahon, "The Symbolics Genera Programming Environment," *IEEE Software*, vol. 4, no. 6, pp. 36-45, November 1987.
26. M. Weiser, L. P. Deutsch, and P. B. Kessler, "UNIX Needs a True Integrated Environment: CASE Closed," Technical Report CSL-89-4, Xerox PARC, 1989. Earlier version published as Toward a Single Milieu, *UNIX Review 6*:11.

27. W. A. Wulf, R. Levin, and S. P. Harbison, *Hydra/C.mmp: An Experimental Computer System*, McGraw-Hill, New York, 1981.
28. *Usenix Workshop on MicroKernels and other Kernel Architectures*, Seattle, WA, 27-28 April 1992.

Author Information

Bill Garrett is a graduate student in the Computer Science Department at the University of Rochester. He received his B.S. from Alfred Unviersity in 1990 and his M.S. from Rochester in 1992. He can be reached c/o the Computer Science Department, University of Rochester, Rochester, NY 14627-0226. His e-mail address is garrett@cs.rochester.edu.

Michael Scott is an Associate Professor of Computer Science at the University of Rochester. He received his Ph.D. from the University of Wisconsin – Madison in 1985. His U.S. mail address is the same as Bill Garrett's. His e-mail address is scott@cs.rochester.edu.

Ricardo Bianchini, Leonidas Kontothanassis, Andrew McCallum, and Bob Wisniewski are graduate students in the Computer Science Department at the University of Rochester. Jeff Thomas is a graduate student in the Computer Science Department at the University of Texas at Austin. Steve Luk is an undergraduate at the University of Rochester, majoring in Computer Science.

A Library Implementation of POSIX Threads under UNIX

Frank Mueller[1] – Florida State University

ABSTRACT

Recently, there has been an effort to specify an IEEE standard for portable operating systems for open systems, called POSIX. One part of it, the POSIX 1003.4a threads extension (Pthreads for short) [12], describes the interface for light-weight threads that rely on shared memory and have a smaller context frame than processes.

This paper describes and evaluates the design and implementation of a library of Pthreads calls that is solely based on UNIX. It shows that a library implementation is feasible and can result in good performance. This work can also be used as a comparison of the performance of other implementations, or as a prototyping, testing, and debugging system in the regular UNIX environment. Finally, some problems with the Pthreads standard are identified.

Introduction

Light-weight threads are independent threads of control within a regular process that share global data (global variables, files, etc.) but maintain their own stack, local variables, and program counter. Threads are referred to as light-weight because their context is smaller than the context of processes. Therefore, context switches between threads may be cheaper than context switches between processes. Furthermore, threads are an adequate model to implement Ada tasks and provide a simple but powerful model for exploiting parallelism in a shared-memory multiprocessor environment. The POSIX threads extension specifies a priority-driven thread model with preemptive scheduling policies, signal handling, and primitives to provide mutual exclusion as well as synchronized waiting. Although the Pthreads draft is not yet a standard and is still being changed through a balloting process, we will refer to the document [12] as the "Pthreads standard". More background on programming with threads is given in [5, 11] as well as in a previous paper describing the early stages of this implementation [17].

This work focuses on the design and implementation issues of POSIX threads (Pthreads) on the Sun SPARC architecture. It describes a true library implementation with a minimal interface to Sun UNIX 4.3 BSD and evaluates its performance.

This article is structured as follows: An overview of previous work in the area precedes the design decisions and their motivations. Then, a brief overview of the Pthreads standard is followed by a more detailed discussion of the design and implementation. Finally, measurements and their evaluations, unresolved problems with the standard, future work, and summary follow.

Related Work

Cthreads, an early implementation of threads, is a coroutine-like extension of the language C. A library implementation was used as a teaching tool by Cooper [7]. This original notion of Cthreads lacked priorities, did not handle signals on a per-thread basis, and supported only non-preemptive scheduling. The first commercial operating system to support threads was the Mach OS [23]. Cooper also provided an implementation of Cthreads based on Mach threads thereby supporting preemption. An early library implementation of prioritized preemptive threads at Brown University [14] supports various architectures including a multiprocessor and handled signals asynchronously. Lately, some commercial operating systems (e.g., LynxOS [9], SunOS [18, 22]) support Pthreads by using a mixture of library and kernel implementation, while others such as Chorus [1] provide more functionality as part of the kernel. An earlier, partial implementation of Pthreads on the library level [19] was used as a base for this project.

Motivation

The Pthreads standard provides a uniform base for multiprocessor shared-memory applications, real-time system environments, and a cheap model for multi-threaded programs on a single processor. The notion of threads can be used to implement Ada tasks or to express parallelism within applications at the level of programming languages. An implementation of Pthreads can be carried out as:

[1]This work was partially funded by the Ada Joint Program Office, through the U.S. Army CECOM and Telos Corp.

- a kernel implementation, where all functionality is part the the operating system kernel;
- a library implementation, where all functionality is part of the user program and can be linked in; or
- a mixture of the above.

A kernel implementation simplifies control over thread operations and signal handling but adds the overhead of entering and leaving the kernel at each call. A library implementation can be more efficient since it does not have to enter the operating system kernel but it complicates signal handling and some thread operations, and it also has to deal with two different scheduler, one for processes (kernel level) and one for threads (library level).

This study discusses the issues of a true library implementation which can be used on a SPARC architecture without specific operating system support for threads. It has been used successfully in an effort to implement an Ada runtime system on top of Pthreads to make the Ada runtime system more portable and to show that the overhead of layering a runtime system on top of Pthreads is not prohibitive.

Pthreads Standard

The Pthreads standard specifies various services that can be provided to support multi-threaded applications. Most of the interface specifications leave many details to the implementation. For example, support for certain functions and the detection of some errors is optional. Therefore, Pthreads-compliant implementations may vary considerably. This implementation supports the following functionality:

- *thread management:* initializing, creating, joining, exiting, and destroying threads;
- *synchronization:* mutual exclusion, condition variables;
- *thread-specific data*;
- *thread priority scheduling:* priority management, preemptive priority scheduling;
- *signals:* signal handlers, asynchronous wait, masking of signals, long jumps;
- *cancellation:* cleanup handlers, different interruptibility states.

The support is currently being extended to include process control.

Design and Implementation

The design of Pthreads has been strongly influenced by constraints of the Pthreads standard, limitations due to the approach of a library implementation, and to some extend by the use of SunOS (UNIX 4.3 BSD) on a SPARC architecture. The machine-dependent part of the implementation consists of about 400 lines of predominantly assembly code. The interface consists of a C library with linkable entry points and can optionally be compiled to generate a language-independent interface. A language interface for Ada has already been designed and tested. Figure 1 illustrates the different software layers of the design.

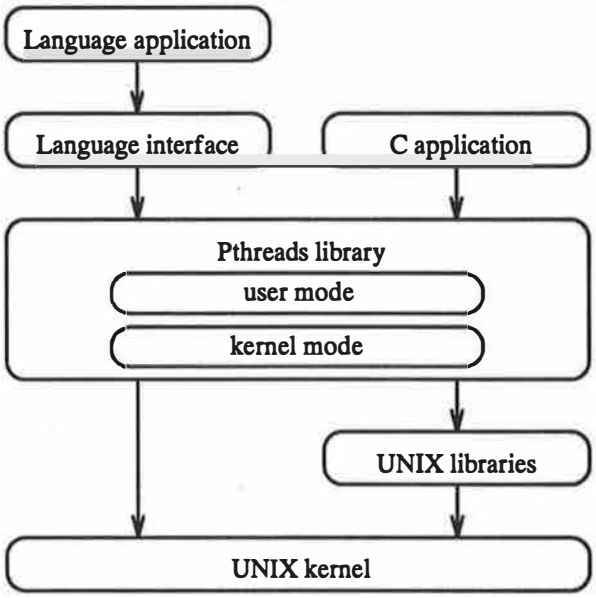


Figure 1: Software Layers

An interface allows programs to use Pthreads services. In case of the programming language C the library routines of Pthreads are immediately available. Any other programming language needs a language interface to the Pthreads library to pass parameters correctly, perform type conversion and other language or compiler-dependent adjustments. The Pthreads library contains a set of routines whose interface and functionality are defined by the Pthreads standard. The code of Pthreads routines partially executes as user code and, within critical sections, operates in the Pthreads kernel mode which guarantees mutual exclusion between threads. The implementation uses a number of UNIX standard library routines and UNIX kernel calls. The design was driven by the following objectives:

- *Preemptability:* Scheduling policies such as round-robin scheduling and asynchronous events (signals) together with priorities can only be supported by a preemptive kernel design.
- *Fast Context Switches:* The context switch is the only means by which control is transferred from one thread to another. A thread's light weight should reduce the context switch overhead.
- *Small Critical Sections:* The time spent in critical sections should be as short as possible. The overhead of entering and leaving critical sections should also be small.
- *No unlimited Stack Growth:* If an asynchronous event arrives while executing an interrupt handler, another handler may be pushed onto the stack and so on *ad infinitum*. A

scheme for handling signals that avoids unlimited stack growth is described below.

- *Few Operating System Calls:* Since calls to the operating systems are time-consuming operations, the use of them should be minimized, especially in time-critical places such as signal handling and context switches.
- *Language-Independent Interface:* The implementation should support the design of an interface to Pthreads with a minimum of dynamic overhead for programming languages other than C.

Pthreads Kernel

Structures allocated by Pthreads must be protected from being modified inconsistently during the handling of asynchronous events (signals). To provide such a protection the library implementation guarantees that critical sections of the library code can only be executed by one thread at a time. The technique which was used to provide mutual exclusion for this implementation is commonly known as a *monolithic monitor* and will be referred to as the library kernel or simply *kernel* in the following.

An alternative to using coarse-gained locking, such as a monolithic monitor, would be to perform fine-grained locking where a different semaphore is associated with each global data structure. The latter approach allows for more concurrency in a multiprocessor environment but more operations need to be performed to guarantee mutual exclusion for each data structure individually. Since this implementation is dedicated to a uniprocessor environment, it was decided to implement a monolithic monitor.

The Pthreads kernel can be entered by setting the kernel flag. Thereafter, any operations are protected so that modifications to thread-internal data structures are guaranteed to be performed in mutual exclusion with other threads. Another flag, the dispatcher flag, indicates whether the dispatcher will be invoked when leaving the Pthreads kernel. The flag is set when a new thread is scheduled or when a signals is received while executing in the Pthreads kernel. To leave the Pthreads kernel, the kernel flag is simply reset if the dispatcher flag was not set; otherwise the dispatcher is invoked which might result in a context switch to another thread. This also allows the implementation to handle signals received from within the kernel as explained below.

Signal Delivery

The delivery of process-level signals to threads is closely coupled with the dispatcher. In particular, signals received while in the kernel are handled differently than signals received while executing instructions outside the kernel although they share a universal signal handler on the process level.

During the initialization of Pthreads a universal signal handler is installed for all maskable UNIX signals. When a signal is caught by the universal handler and the kernel flag is not set, the kernel is entered by setting the kernel flag, all signals are enabled, and a routine is called which first directs the signal at the appropriate thread and then calls the dispatcher. The control might not immediately be transferred back to the same thread if the signal made a higher priority thread eligible to run.

When a signal is caught while in the kernel, the received signal is logged and its handling is deferred until the dispatcher is called. The control is then immediately transferred back to the interruption point by returning from the universal signal handler which also enables signals at the process level again.

Thread States

A thread may be blocked waiting for some event, ready to execute (but not chosen yet by the scheduling policy to be dispatched), running (dispatched), or terminated (cannot be scheduled anymore). Furthermore, a thread may be detached in conjunction with any of the above states.

After a detached thread terminates or after a terminated thread is detached, any memory associated with the thread can be reclaimed and the thread may not be referenced any longer.

The Dispatcher

Under normal circumstances, a call to the dispatcher will select the next thread eligible to run from the set of ready threads according to the scheduling policy. If the selected thread differs from the thread currently running a context switch has to be performed. A thread context switch on the SPARC consists of

- saving non-scratch registers of the current thread which is accomplished on the Sun SPARC by a trap into the UNIX kernel to flush the set of active register windows onto the stack (`ST_FLUSH_WINDOWS`),
- loading the frame pointer with the top of the thread's stack,
- loading UNIX' global error number with the thread's error number,
- loading non-scratch registers and switching to the frame of the new thread by executing a restore instruction, and
- transferring control to the new thread.

On the SPARC, the only registers changed during a thread's context switch are those describing the local state contained in the register windows (ins/outs and locals). Any global state such as global registers, floating point registers, and the status word never need to be updated during a context switch because these registers are either considered to be scratch registers (across explicit calls to Pthreads routines) or are saved by the UNIX (when a signal is delivered). When a thread which was not interrupted by a signal is dispatched, the context is logically switched to the local state of the new thread (see Figure 2). Before the control is transferred

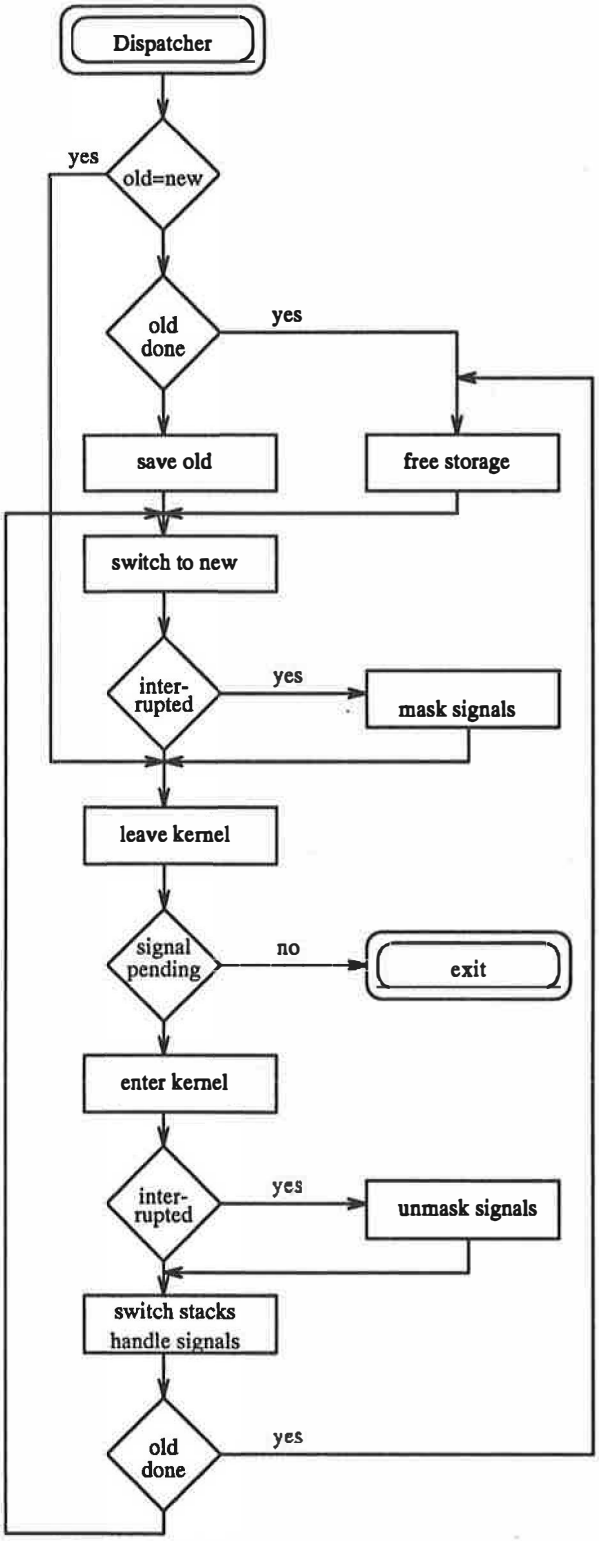


Figure 2: Flowchart of the Dispatcher: Switch Context from old to new Thread

to the new thread, the kernel and dispatcher flags are cleared and it is checked whether signals were caught while in the kernel. If no signals were caught the control can be transferred to the new thread; otherwise the signals will be handled as explained below and another attempt to dispatch a thread will follow. Since the handling of signals might change the thread to be dispatched next, the context switch has to be restarted.

When an interrupted thread is chosen to be dispatched, the universal signal handler will still be pending on top of the thread's stack. Therefore, the dispatcher disables all signals before initiating the context switch. When the thread regains control it will return from the universal signal handler, enable all signals again, and return to the UNIX interrupt frame which will restore the global state (global registers, floating point registers, and the the status word). It is essential to disable signals before switching to the context of an interrupted thread to avoid unbounded stack growth. Otherwise the universal signal handler could be interrupted by yet another instance of the universal signal handler (and so on) before the thread can return from the first instance of the handler.

Signal Handling

Process level signals are deferred until the dispatcher is called if they were caught while in the Pthreads kernel. Otherwise, they are handled immediately. Signal handling determines the receiving thread and the action to be taken for the signal. The recipient is determined according to the so-called *signal delivery model* which describes when a thread receives a signal and how conflicts between multiple threads are resolved. This implementation uses the following conflict resolution (beginning with the highest precedence):

1. If the signal is specifically directed at a thread, this thread is the recipient; else
2. if the signal is delivered synchronously, direct it at the thread which caused it; else
3. if the signal was caused by a timer expiration, direct it at the thread which armed the timer; else
4. if the signal was caused by an I/O completion, direct it at the thread which requested I/O; else
5. if any thread has the signal unmasked, direct it at such a thread; else
6. pend the signal on the process level until a thread becomes eligible to receive it.

The choice of an arbitrary thread is sufficient in step 5 to comply with the Pthreads standard. This implementation performs a linear search of a list of all threads until either all threads are exhausted in the search or a thread is found which has the signal unmasked. (The routine `sigwait` is just another case where the signal is unmasked).

If a thread is selected as the recipient of a signal an action will be selected as follows (beginning with the highest precedence):

1. If the thread masked the signal, pend the signal on the thread; else

2. if the signal is the alarm signal and was caused by a timer expiration, the selected thread either becomes ready if it was suspended or it is position at the tail of the ready queue if the timer expiration was furthermore caused by time-slicing; else
3. if the thread suspended in a call to `sigwait`, the thread becomes ready and signals specified in the call to `sigwait` are masked for the thread; else
4. if a handler has been registered for the signal, a *fake call* will be installed for the selected thread, signals are masked according to the mask specified in `sigaction`, and the thread becomes ready; else
5. if the signal is the cancellation signal (see section "Thread Cancellation"), a *fake call* to `pthread_exit` is pushed onto the threads stack and the thread becomes ready; else
6. if the action defined on the signal is to ignore the signal, take no action and discard the signal; else
7. if the action defined on the signal is the default action, perform the default action on the process.

Fake Calls

Thread signal handlers (user handlers) installed by a call to `sigaction` are invoked through a mechanism called *fake call.* A fake call pushes a frame on top of a thread's stack and sets up the frame to act as if a function had been called explicitly by the thread.

The use of fake calls as a mechanism to invoke user handlers is motivated by the constraint that user handlers have to execute at the priority level of the corresponding thread. Rather than making an explicit call to the user handler when a process signal is received, the execution of the user handler has to be deferred until the receiving thread is dispatched. This is enforced through the use of fake calls.

Figure 3 illustrates the mechanism of fake calls. User code is interrupted by a signal causing the operating system to create a new frame which saves the state at the interruption point and invoke Pthreads' universal signal handler which calls the dispatcher. The dispatcher changes to the temporary stack (as indicated by an arrow) but remains active. While executing on the temporary stack the signal is directed to a thread (in this case the interrupted thread), and a new frame, a wrapper, is created on top of the thread's stack (indicated by another arrow). The program counter and stack pointer of the thread have to be updated to reflect the new frame of the fake call, which will execute the wrapper when the thread regains control. The wrapper takes the following actions:

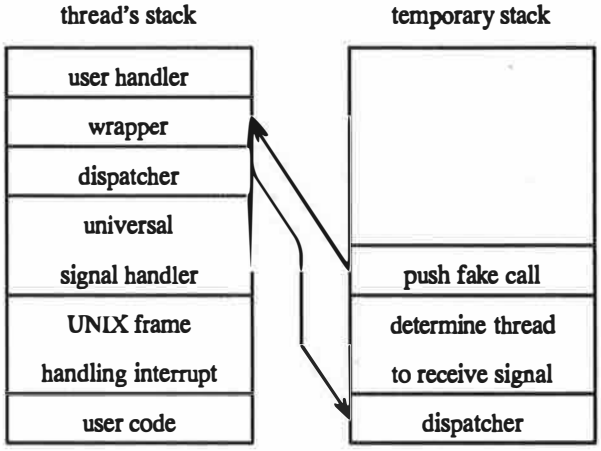


Figure 3: Fake Call onto same Thread

- If the user handler interrupted a conditional wait, the mutex is reacquired and the conditional wait terminated;
- the thread's error number is saved;
- the user handler is called;
- the thread's error number is restored;
- the requested per-thread signal mask is restored and pending signals on the thread and process are handled if now enabled;
- the control is either transferred back to the interruption point or to an instruction whose address can optionally be specified by the user handler.

The ability to redirect control to some specified address is a feature not required by the Pthreads standard but rather left open as implementation defined. Nevertheless, this feature is essential for the Ada runtime system: When a synchronous signal is received, one needs to return from the user handler and restore the previous frame before propagating the exception corresponding to the signal. The Ada runtime system also makes use of the signal code which, in some cases, distinguishes

Interruptibility		Action
State	Type	
disabled	any	SIGCANCEL pends on thread until cancellation is enabled
enabled	controlled	SIGCANCEL pends on thread until interruption point is reached
enabled	asynchronous	Cancellation is acted upon immediately

Table 1: Action taken upon Cancellation Request

between different causes of the same synchronous signal.

Thread Cancellation

A thread may be cancelled by calling `pthread_cancel`. The cancellation is handled as a request for sending a special (internal) signal SIGCANCEL to a thread. Depending on the interruptibility state of the receiving thread, an action will be taken upon a cancellation request according to Table 1.

Interruption points are functions defined in the Pthreads interface which may suspend a thread indefinitely (e.g., conditional waits) with the exception of locking a mutex. Locking a mutex should not be an interruption point to allow for efficient implementations. An interruption point can also be created by calling `pthread_testintr`.

If a cancellation request is acted upon, the interruptibility state of the receiving thread is changed to *disabled*, all other signals are disabled for this thread, and a fake call to `pthread_exit` is pushed onto the thread's stack.

UNIX Interface

To maximize the performance of a true library implementation, calls to the operating system kernel need to be minimized. The overhead associated with entering and leaving the UNIX kernel makes kernel calls expensive operations. This implementation makes use of about 20 UNIX services most of which are used for initialization of the Pthreads library and a few other non-time-critical stages. However, there are a few exceptions.

- When a context switch is performed on the SPARC the register windows of the current thread are flushed via a system trap instruction. The register windows of the new thread will be loaded when the restore instruction is executed which causes a window underflow trap. These two traps consume most of the time required for a context switch and are inherent to any context switch on the SPARC.
- Thread creation/termination involves allocation/deallocation of heap space which sporadically may result in kernel calls to `sbrk`. This could be avoided in most cases by preallocating a pool of thread control blocks and stacks. Thus, dynamic memory allocation would only be performed when the pool space is exhausted at creation time.
- It is most crucial to minimize the use of kernel calls when signals are caught or handled; in particular, calls to `sigsetmask` need to be minimized and signals should be blocked for the shortest interval possible to avoid the loss of signals at the UNIX process level. This implementation uses two calls to `sigsetmask` for each signal received by the process.

Synchronization

The Pthreads standard specifies a "mutex" object, a data structure for mutually exclusive access of shared data structures and condition variables for synchronization between threads. Other synchronization methods such as counting semaphores [3] can be easily implemented on top of these primitives [17].

A thread may acquire (lock) an unlocked mutex. Thereafter, mutual exclusion between threads in the same process is guaranteed, until the mutex is unlocked provided that other threads guard critical sections by the same mutex. If a thread tries to lock a mutex which is already locked, the thread suspends. If a thread unlocks a mutex and other threads are waiting on the mutex, the waiting thread with the highest priority will acquire the mutex. To simplify implementations, a thread cannot be cancelled while in controlled interruptibility when it suspends due to mutex contention to guarantee a deterministic state of the mutex in cleanup handlers.

A mutex should only be locked for the shortest possible time to minimize contention. For example, one should protect access to data structures shared between threads by mutexes. But one should not lock a mutex, perform an action which might cause suspension of the thread, and then unlock the mutex, since contention is likely to occur while the thread holding the mutex suspends.

To allow synchronization between threads and suspension over a longer (possibly unbounded) time interval, the standard introduces condition variables. A mutex and a predicate based on shared data are associated with a condition variable. When a thread wants to synchronize with another thread, it locks the mutex, tests the predicate and, if the predicate evaluates to false, suspends on a conditional wait. When the thread is reactivated, it reevaluates the predicate and so on until the predicate becomes true.

The reevaluation of the predicate is essential since spurious wakeups on a multiprocessor and wakeups due to asynchronous events may cause the thread to resume execution while the predicate still evaluates to false.

When a thread enters a conditional wait with the associated mutex locked, the mutex is unlocked atomically with the suspension of the thread. Similarly, the mutex is atomically relocked when thread resumes execution. Thus a mutex is always in a known state even when signals interrupt a conditional wait since the mutex will be reacquired before any interrupt handler starts executing.

A condition variable is typically signaled by a thread after the thread changes the state of some shared data allowing the associated predicate to evaluate to true. When a condition variable is signaled, at least one of the threads blocked on it become ready. If more than one thread is blocked

on a condition variable, the thread with the highest priority will become ready. In particular on multiprocessors, an implementation that allows multiple threads become unblocked on signaling a condition variable may be more efficient.

Mutexes should be implemented to provide mutual exclusion in the most efficient way possible. Ideally, a simple test-and-set instruction should be sufficient. Unfortunately, this would result in several deficiencies:

- The standard also specifies more complex optional protocols such as priority inheritance to avoid priority inversion (see section "Priority Inversion: Inheritance and Ceilings"). Priority inheritance requires that the ownership of the mutex be recorded atomically with the locking operation.
- Hardware implementations of test-and-set instructions often perform worse than restartable atomic instruction sequences for mutual exclusion on a uniprocessor [4].

The priority inheritance protocol requires that if a high priority thread suspends on a mutex due to contention with a low priority thread which holds the mutex, the low priority thread inherits the high priority until it unlocks the mutex. Thus, the ownership association of a mutex allows the high priority thread to boost the priority of the thread holding the mutex. This will be discussed in more detail later. But first, several implementation options for recording the ownership of a mutex atomically with locking the mutex shall be considered.

A restartable atomic sequence is guaranteed to be atomic by augmenting the signal handler. If such a sequence was interrupted by the signal handler, the atomic sequence is restarted in the signal handler; otherwise no action in taken. For the implementation of the mutex lock, it is thereby guaranteed that there be an owner associated with every locked mutex at any given time.

This scheme does not extend to multiprocessors. Rather, test-and-set instructions become essential as they are the only means to guarantee atomic updates of memory. But restarable atomic sequences can be used to record ownership in conjunction with a test-and-set instruction on multiprocessors by letting the contending thread spin until the bounded interval between locking the mutex and setting the owner must have passed for the acquiring thread [15].

On the SPARC, the test-and-set instruction is about as fast as a restartable atomic sequence [4]. It was therefore decided to use the test-and-set instruction for mutual exclusion but executed it inside a restartable atomic sequence which also included the recording of the ownership. Such a sequence consists of 7 instructions in our implementation (see Figure 4), two instruction more than required by SunOS 5.0 [15]. Sun reserves a hardware register to contain the current thread ID at any time which saves an address calculation and a load required in our implementation. Since this hardware register is reserved for internal use by the SPARC Application Binary Interface [21], such properties cannot be guaranteed by our implementation for any register without changing the operating system kernel.

```
ldstub  [%o0+mutex_lock],%o1
tst     %o1
bne     mutex_locked
sethi   %hi(_kern),%o1
or      %o1,%lo(_kern),%o1
ld      [%o1+pthread_self],%o1
st      %o1,[%o0+mutex_owner]
```

Figure 4: Atomic Sequence to Lock a Mutex and Record the Owner

An additional atomic instruction besides the test-and-set instruction would have avoided these problems: Consider a *compare-and-swap*) which atomically tests some memory word and sets it to the value of a specified register if the memory location contained zero. Let the condition flags also be set by the testing. Then this instruction could be used to record ownership instead of the restartable atomic sequence. Such an approach removes the overhead induced on signal handlers by atomic sequences. But the compare-and-swap instruction would need two more cycles to execute than the test-and-set to perform the comparison and decide whether to update the memory word. This does not seem critical though since a test-and-set instruction will always be followed by a test and a conditional branch instruction to check on the success of the operation. The encoding in 32 bits is the same for both the test-and-set and compare-and-swap instruction since a memory location and a register will be specified in both cases. Therefore, a compare-and-swap instruction should be provided in the instruction set of any processor.

Ada Interface and Binding

A system-level interface between the Pthreads C library and the language Ada has been implemented. This interface has been used to implement an Ada runtime system which is able to map Ada tasks onto threads due to the similarity of their properties. The runtime system can be easily ported to other systems that support Pthreads except for a few implementation-dependent features of Pthreads (e.g., use of signal context record). An Ada binding for Pthreads (user-level interface), on the other hand, is more complex than a bare language interface. Several services provided by Pthreads interfere with the Ada language definition, in particular the handling of signals. We are currently engaged in an effort to define a suitable subset of Pthreads operations as a safe Ada binding [10].

It is suggested by the Pthreads standard that several Pthreads routines be implemented as C macros. Unfortunately, this is a severe limitation to the language-independent approach taken otherwise.

Most notably, cleanup handlers are suggested to be implemented as a pair of macros: `pthread_cleanup_push` opens a new lexical scope, declares a cleanup structure automatically, and links it to a thread-specific cleanup stack. `pthread_cleanup_pop` restores the previous state of the cleanup stack and closes the lexical scope. Since this implementation depends on the creation of lexical scopes it cannot be incorporated as a function call into a language interface. Another layer would have to be included to embed the macro call into a regular C function. Furthermore, the current cleanup structure could no longer be allocated as a local variable within the new lexical scope but would have to become a global variable. Finally, to guarantee that the two operations occur as a pair in the same lexical compiler support would be needed.

It was decided to avoid C macros for interface implementations in general. This trades the overhead of function calls otherwise not needed by C applications for the generality and language-independence of the interface. Such an approach seemed favorable over implementation-specific solutions.

Measurements and Evaluation

The Pthreads standard suggests a set of performance metrics based on the set of routines defined in the interface. Table 2 shows selected measurements used in previous studies. The measurements for our implementation were taken on a Sun SPARC 1+ (column 3) and on a Sun SPARC IPX (column 4) under SunOS 4.1 using dual loop timing analysis. Some measurements are compared to those reported for SunOS [18] (column 2) taken on a Sun SPARC 1+. Others are compared to the results reported for a pre-release of LynxOS (column 5) taken on a Sun SPARC IPX.

The benefit of a library implementation is indicated by the fact that to enter and exit the Pthreads kernel is considerably faster than to enter and exit the UNIX kernel. (The latter metric was obtained by timing a `getpid` call.) This is still true for the comparison with Lynx although their performance shows some improvement over traditional UNIX kernels.

The metrics included a pair of mutex lock and unlock operations, first under the assumption the a mutex is requested while unlocked, and second the interval between an unlock by thread A and the return from a lock operation by thread B (which was suspended while A held the mutex). Mutexes are a mechanism designed for fine-grain locking and should consequently only be held for a short time. A thread should therefore seldom suspend on a mutex lock. Thus, it should be attempted to maximize the performance of mutex operations without contention. Semaphore synchronization refers to one Dijkstra *P* operation plus one *V* operation and were implemented on top of mutexes and condition variables [17]. Neither Lynx operations for synchronization nor Sun's "unbound thread synchronization" via semaphores is reported to perform as well as ours.

Further measurements were taken for creating a new thread (excluding the context switch time). The thread control block and stack were pre-cached in a memory pool to avoid dynamic memory allocation.

Performance Metric	Time[μsec]			
	Sparc 1+		Sparc IPX	
	Sun	Ours		Lynx
enter and exit Pthreads kernel			0.4	7.5
enter and exit UNIX kernel			18	
mutex lock/unlock, no contention			1	5
mutex lock/unlock, contention			51	
semaphore synchronization	158	101	55	75
thread create, no context switch	56	25	12	
setjmp/longjmp pair	59	49	29	
thread context switch (yield)			37	38
UNIX process context switch			123	41
thread signal handler (internal)			52	
thread signal handler (external)			250	
UNIX signal handler			154	

Table 2: Performance Metrics

Sun's "unbounded thread creation" corresponds to this test as it makes the same assumptions. Comparing the measurements, thread creation of this implementation seems to be faster than Sun's.

The performance of a pair of `setjmp` and `longjmp` operations gives a lower bound on the overhead of a context switch but a true context switch involves some additional overhead as indicated by the measurements. Again, this implementation exceeds the speed reported by Sun but matches Lynx'. Little tuning is possible for the context switch on the SPARC since most of the time is spent in the kernel traps to save and restore registers. Also notice that UNIX process context switches are considerably slower than thread context switches. For LynxOS the performance of context switches hardly differs between processes and threads. (The UNIX process context switch time was measured by timing the execution of two alternating processes which activate each other by exchanging signals minus the time required for process signal delivery.)

The measurements taken for signal handling reflect the time it takes from sending a signal until the signal is received. Since this implementation is build on top of the somewhat slow signal handlers provided by UNIX, external signal handling for threads (i.e., signals directed at the process and demultiplexed to threads) is a time-consuming operation. The performance of internal signal handling (i.e., signals directed at a thread for within the process) indicates that a faster implementation might be possible if the operating system kernel was redesigned. This suggests either that threads be implemented as part of the operating system such that signals can be handled within the kernel or that the kernel/user interface allows the kernel to send the signal to the correct user thread directly [16].

Overall, this implementation seems to match and in some cases exceed the performance reported by commercial implementations.

Perverted Scheduling: Testing and Debugging

Debugging on multiprocessors is typically more complex, more expensive (since a whole set of multiprocessors might be blocked), and errors might not always be reproducible. This library implementation can be helpful to detect and analyze errors in a uniprocessor environment before an application is tested on a multiprocessor. Two types of errors can be distinguished, *serial* errors which occur in uniprocessor environments and *parallel* errors which are inherent to parallel execution. Debugging the former type of errors is well understood. But errors of the latter type are often hard to detect. This implementation of Pthreads has been extended for debugging purposes to optionally provide *perverted scheduling*, a set of scheduling policies which simulate parallel execution on multiprocessors. The following set of perverted scheduling polices has been implemented:

- *Mutex Switch:* On each successful locking of a mutex, a context switch is forced by repositioning the current thread at the tail of its priority queue. The thread at the head of the ready queue executes next.
- *Round-Robin Ordered Switch:* On leaving the Pthreads kernel, a context switch is forced by repositioning the current thread at the tail of the lowest priority queue. The thread at the head of the ready queue executes next.
- *Random Switch:* On leaving the Pthreads kernel, a context switch is forced if the next binary random number produced by some pseudo random-number generator is true. In this case, the current thread is repositioned at the tail of the lowest priority queue and the next thread to execute is selected by randomly choosing a thread from the ready queue.

The above policies may not always conform with priority scheduling as defined in the Pthreads standard. In fact, for the latter two a lower priority thread may execute while a higher priority thread is ready. But on a multiprocessor, the execution of high and low priority threads might occur in parallel. By alternately executing high and low priority threads, this implementation tries to simulate parallel execution using concurrency.

Perverted scheduling policies are easier to deal with than time-sliced round-robin scheduling since in the time-slicing case context switches are caused by timer expirations. In multiprogramming environments timer expirations may vary according to a processor load and in a debugging environment timer expirations may further vary depending on debugging actions. Thus, errors which occur during time-sliced round-robin scheduling may not be reproducible.

The perverted scheduling policies have been used to test the robustness of our implementation of an Ada runtime system based on Pthreads. Several errors were detected which did not show up under the FIFO scheduling policy. But none of the errors were inherent to multiprocessors, they could have occurred on a uniprocessor as well with a different (and legal) ordering of execution of threads. Varying the initialization of random number generators for the random switch policy also proved to be a simple but powerful way to influence the ordering of threads during execution. Still, more experience has to be gained to understand all the benefits of perverted scheduling.

Open Problems

Several problems regarding the POSIX standard have to be resolved. One problem, the use of C macros as discussed in a previous section, suggests that greater emphasis should be placed on language-independence.

Non-Blocking Kernel Calls

UNIX does not provide non-blocking equivalents of some blocking system calls, for example for the interface to directories in the file system. Other non-blocking interfaces for I/O, for example, do not provide the correct semantics with regard to POSIX when interrupted by signals.

Marsh and Scott [16] have made suggestions to overcome some problems associated with user-level threads by defining a generic interface between operating system kernel and user level. This interface provides fast communication between the kernel and user-level activity. For example, when issuing non-blocking I/O request the kernel associates the request with a user-provided datum (the calling thread) such that the user-level thread scheduler can be notified of the I/O completion in conjunction with this datum. This obviates signal demultiplexing at the user level which should increase the response to asynchronous events considerably without unduly complicating the operating system kernel.

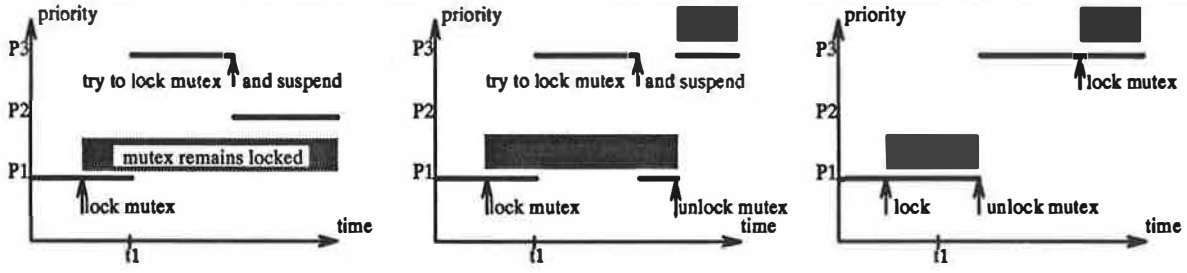
Priority Inversion: Inheritance and Ceilings

Combining priorities and critical sections may cause priority inversion, a situation where a higher priority thread cannot preempt the lower priority thread executing a critical section. Priority inversion may result in unacceptably long delays within multithreaded operating system kernels (microkernels) [8] and user applications. Furthermore, it might not be possible to guarantee timing constraints of real-time systems.

Consider the example in Figure 5(a). A solid line indicates that a thread is executing and a grey box over a thread shows that the thread holds a mutex. A low priority thread P1 locks a mutex and is preempted at *t1* by a high priority thread P3. Thread P3 then tries to lock the same mutex and blocks since the mutex is held by P1. At *t1*, a medium priority thread P2 also has become ready and starts to execute when P3 suspends. Thread P2 does not contribute to the progress of P3 since P3 will not resume its execution until P1 releases the

	Inheritance Protocol	Ceiling Protocol (via SRP)
Boost Priority	of thread holding mutex when a high priority thread suspends on the mutex lock	of the current thread when the mutex is acquired
Boost Prio Level	set to max(own prio, prio of contending threads) by the contending thread	set to prio ceiling of mutex by locking thread
Lower Priority	on unlocking mutex	on unlocking mutex
Lower Prio Level	set to max(own prio, prio of contending threads of other mutexes remaining locked)	set to level before acquiring the mutex
Implementation	linear search of locked mutexes (unlock)	push/pop of ceiling values (stack)
Adaptability	adjusts dynamically to prio level of threads at lock static, prio ceiling set to at least	max(prio of threads locking mutex) at initialization
Bound on Inversion	sum of longest critical section of lower prio threads	tighter: longest critical section of lower prio threads

Table 3: Properties of Synchronization Protocols



(a) no protocol (b) inheritance protocol (c) ceiling protocol

Figure 5: Dealing with Priority Inversion

mutex. Thus, the priorities of P2 and P3 are inverted, the lower priority thread P2 continues its execution without giving the higher priority thread P3 any chance to regain control.

Several protocols have been suggested to overcome priority inversion. The Pthreads standard specifies two protocols, priority inheritance [20] and priority ceiling emulation which can be implemented efficiently using SRP (stack resource policy [2]). A short comparison between the two protocols is given in Table 3. While the implementation of ceilings via SRP is considerably more efficient, priority inheritance can adjust to dynamic changes of priorities which cannot be anticipated and may perform better when contention is rare. For the ceiling protocol, the priority ceiling of a mutex has to be initialized at compile time to a at least the maximum priority of all threads that may lock this mutex. Thus, priority ceilings are associated with the synchronization object (mutex) while priority inheritance is concerned with the priority of threads.

Consider the example in Figure 5(b) with the inheritance protocol. P1 inherits P3's priority when P3 tries to lock the mutex. Thus, P1 runs until it unlocks the mutex and lowers its priority to the original level. P3 then continues to execute since it has the highest priority and can now acquire the mutex. Priority inversion is avoided since P2 does not get to run.

With the ceiling protocol in Figure 5(c), the priority ceiling of the mutex matches (or even exceeds) P3's priority since P3 is the highest priority thread locking the mutex. Thus, when P1 locks the mutex, its priority is raised to the ceiling level. When P1 unlocks the mutex, its priority is lowered to the original level. Although P3 has become ready at *t1* it can only preempt P1 when the mutex becomes unlocked due to the priority ceiling. Later on, P3 locks the mutex and its priority is boosted to the ceiling value. Priority inversion is avoided since P2 never runs. Notice that this protocol tends to require fewer context switches than the inheritance protocol and mutexes are locked for a shorter time.

Several observations regarding the Pthreads standard were made when trying to implement the forementioned protocols:

- The inheritance protocol and the ceiling protocol can be implemented independently. But the Pthreads standard allows ceilings only in the presence of inheritance, which seems to be too restrictive and should be changed.
- The ceiling protocol can be implemented much more efficiently (using a stack [2]) if critical sections are nested properly. Thus, if mutexes are unlocked in a different order than they were locked, the behavior should be undefined for at least the ceiling protocol. Also, if the ceiling of a mutex is not set to the level of the highest priority thread which may lock it, the effect should be undefined. The Standard does not specify such restrictions.
- The implementation of different protocols compromises efficiency. There is only one routine for locking and one for unlocking a mutex defined in the interface. The different protocols are identified by attributes. A simple mutex lock (no protocol) could have been implemented with a test-and-set instruction. But it now requires an additional check of the attributes. It seems preferable to provide different interfaces for each protocol since the actions taken in each case vary considerably.
- The relation of priority scheduling and lowering a thread's priority on unlocking the mutex is ambiguous. It is not clear if a thread will be placed at the tail of its priority level queue as required by the priority scheduling policy or at the head.
  The latter approach seems preferable since a priority boost effects a thread only temporarily. It is not the choice of the thread to change its priority. Rather, the thread is forced into a higher priority. Consequently, neither should any other thread at the same priority level be scheduled instead of the current thread when the priority is reset to the initial level due to the ceiling protocol, nor should the effected thread by penalized for boosting its priority. Furthermore, context switches can potentially be avoided.
- The protocols for inheritance and ceiling do not mix well. In particular, if critical sections with different protocols were nested, the implementation of the ceiling protocol would degrade to that of the inheritance protocol since priority levels would not follow the stack principle (LIFO) anymore. If the ceiling

#	Action	*Pi*	*Pc*	Comment
1	lock(inht)	0	0	no contention for *inht*
2	lock(ceil)	1	1	*ceil* has prio ceiling 1
3	...	2	2	contention for *inht*, inherit prio 2
4	unlock(ceil)	2	0	protocol divergence
5	unlock(inht)	0	0	

Table 4: Mixing Inheritance and Ceiling Protocol

protocol is to be implemented using a stack, the nesting of critical sections using the different protocols for ceiling and inheritance has to be prohibited.

The example in Table 4 illustrates the last point. Consider mutex *inht* with inheritance protocol and mutex *ceil* with ceiling protocol. The priority of the thread using inheritance protocol *Pi* differs from the usage of the ceiling protocol *Pc* in step 4. If the ceiling protocol was implemented as a stack, it would restore the priority prior to locking mutex *ceil*. But this leads to priority inversion for mutex *inht*. If, on the other hand, a linear search was performed on an unlock regardless of protocols, the priority would remain boosted until step 5 and unbounded inversion could be avoided. Thus, the linear search of the inheritance protocol would have to be performed for the ceiling protocol as well if the protocols were mixed.

Future Work

The current status of the implementation still lacks shared mutexes and condition variables which can be used across processes. Such objects could either be implemented on top of existing interprocess communication primitives or by allocating a mutex object in a shared data space. The latter approach should achieve better performance. Nevertheless, enforcing mutex protocols across process boundaries, for example to inherit priorities, seems inefficient in a library implementation since the libraries of the two processes would have to communicate somehow.

It may sometimes be useful to create a new thread but defer its activation, also referred to as lazy thread creation. If threads were used within medium and fine-grain models of parallelism, thousands of threads might be in existence at the same time. Clearly, system resources such as stack space will not suffice for all threads. It may therefore be desirable to create a thread but delay its activation including resource allocation until the thread is "needed" by some other thread, for example due to synchronization. An attribute passed at creation time could indicate that the activation is to be deferred.

The current implementation allocates heap space for the stack and thread control block (TCB) at creation time. This accounts for about 70% of the thread creation time. Thus, thread creation could be sped up considerably if a memory pool for TCB and stack was established as was done by other thread implementors.

A major obstacle to the use of threads is to make C libraries reentrant for threads. Several library calls use global state information, some interfaces are non-reentrant, macros have to be modified, and interruption due to signals has to be considered without sacrificing much performance [13]. This issue has not been addressed yet to supplement our implementation with a thread-safe C library among others.

A programming environment for threads should also provide debugging facilities with support for multi-threading [6]. Information could be extracted from the thread control block and made available to the user. Context switches could become visible to the user. For example, when a context switch is about to occur, the user could choose whether to continue debugging after the suspension point of a thread or whether to change into the context of another thread. In addition, separate debugging windows could be allocated for each thread within a process.

The implementation could be extended to support multiprocessors. Several changes would have to be made to the Pthreads kernel. Most notably, the monolithic monitor would have to be replaced by fine-grain locking of shared data structures to minimize contention between different processors while operating in the kernel mode. Otherwise, the advantages of a multiprocessor could not be fully exploited.

Conclusion

It was shown that a true library implementation of Pthreads is possible and feasible. The discussed implementation supports preemptability, fast context switches between threads, small critical sections, avoids unlimited stack growth, uses few operating system calls, and provides a language-independent interface. The implementation seems to exceed the performance of other, kernel-supported implementations, and has been used successfully as a base to implement the tasking portion of an Ada runtime system which passes validation tests for tasking.

The overhead of separate signal handling for each thread complicates the the design and implementation considerably. Some of the advantages of light-weight threads may have been lost due to the requirements of Pthreads. Furthermore, several problems related to language-independence and mutex protocols in particular have to be resolved in future drafts of the standard. It remains to be seen if the Pthreads standard will gain wide acceptance under these circumstances.

Acknowledgments

I would like to thank the following people: Ted Giering and Pratit Santiprabhob for their valuable help and cooperation during the design. Ted Baker for his suggestion for early locking of mutexes for condition variables and his comments about mutex protocols. Viresh Rustagi for the implementation of asynchronous I/O and round-robin scheduling. Bill Gallmeister for providing comparative metrics for LynxOS.

Availability

The source code of Pthreads is available for non-commercial use via anonymous ftp from `ftp.cs.fsu.edu` (128.186.121.27) in the file `/pub/PART/pthreads.tar.Z` – other material such as related publications can be found in the same directory.

Bibliography

[1] Francois Armand, Frederic Herrmann, Jim Lipkis, and Mark Rozier. Multi-threaded processes in CHORUS/MIX. In *Proceedings of EEUG Conference*, pages 1-13, Spring 1990.

[2] T.P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67-99, March 1991.

[3] M. Ben-Ari. *Principles of Concurrent and Distributed Programming*

[4] Brian N. Bershad, David D. Rerdell, and John R. Ellis. Fast mutual exclusion for uniprocessors. In *Proceedings of the Annual Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 223-233, October 1992.

[5] A. D. Birrell. An introduction to programming with threads. Research Report 35, DEC Systems Research Center, 1989.

[6] Deborah Caswell and David Black. Implementing a mach debugger for multithreaded applications. In *Proceedings of the USENIX Conference*, pages 25-40, Winter 1990.

[7] E. Cooper and R. Draves. C threads. TR CMU-CS-88-154, Carnegie Mellon University, Dept. of CS, 1988.

[8] J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams. Beyond multiprocessing ... multithreading the SunOSkernel. In *Proceedings of the USENIX Conference*, pages 11-18, Summer 1992.

[9] Bill O. Gallmeister and Chris Lanier. Early experience with POSIX 1003.4and POSIX 1003.4a. In *Proceedings of the IEEE Symposium on Real-Time Systems*, pages 190-198, 1991.

[10] E.W. Giering and T.P. Baker. POSIX/Adareal-time bindings: Description of work in progress. In *Proceedings of the Ninth Annual Washington Ada Symposium.* ACM, July 1992.

[11] E.W. Giering and T.P. Baker. Using POSIX-threads to implement Adatasking: Description of work in progress. In *TRI-Ada '92 Proceedings*, pages 218-529. ACM, 1992.

[12] IEEE. *Threads Extension for Portable Operating Systems (Draft 6)*, February 1992. P1003.4a/D6.

[13] Michael B. Jones. Bringing the c libraries with us into a multi-threaded future. In *Proceedings of the USENIX Conference*, pages 81-91, Winter 1991.

[14] Thomas W. Doeppner Jr. Threads – a system for the support of concurrent programming. TR CS-87-11, Brown University, Dept. of CS, 1987.

[15] S. Khanna, M. Sebree, and J. Zolnowsky. Realtime scheduling in SunOS 5.0. In *Proceedings of the USENIX Conference*, pages 375-390, Winter 1992.

[16] Brian D. Marsh, Michael L Scott, Thomas J LeBlanc, and Evangelos P. Markatos. First-calss user-level threads. In *Symposium on Operating Systems Principles*, pages 110-121, October 1991.

[17] Frank Mueller. Implementing POSIXthreads under UNIX: Description of work in progress. In *Proceedings of the Second Software Engineering Research Forum*, pages 253-261, November 1992.

[18] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, and M. Weeks. SunOSmulti-thread architecture. In *Proceedings of the USENIX Conference*, pages 65-80, Winter 1991.

[19] Ganesh Rangarajan. A library implementation of POSIXthreads. Master's Project Report, Florida State University Department of Computer Science, July 1991.

[20] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175-1185, September 1990.

[21] Inc. SPARC International. *The SPARC Architecture Manual: Version 8.* Prentice Hall, Englewood Cliffs, New Jersey, 1992.

[22] D. Stein and D. Shah. Implementing light-weight threads. In *Proceedings of the USENIX Conference*, pages 1-10, Summer 1992.

[23] A. Tevanian, R. F. Rashid, D. B Golub, D. L. Black, E. Cooper, and M. W. Young. MACHthreads and the UNIXkernel: The battle for control. In *Proceedings of the USENIX Conference*, pages 185-197, Summer 1987.

Author Information

Frank Mueller is currently pursuing his Ph.D. in Computer Science at Florida State University. His interests as a student and research assistant include compilers, computer architecture, Ada, real-time systems, and concurrent programming. His U.S. mail address is Florida State University; Department of Computer Science, B173; Tallahassee, FL 32306-4019. His e-mail address is mueller@cs.fsu.edu.

Hello World or Καλημέρα κόσμε or こんにちは 世界

Rob Pike – AT&T Bell Laboratories
Ken Thompson – AT&T Bell Laboratories

ABSTRACT

Plan 9 from Bell Labs has recently been converted from ASCII to an ASCII-compatible variant of Unicode, a 16-bit character set. In this paper we explain the reasons for the change, describe the character set and representation we chose, and present the programming models and software changes that support the new text format. Although we stopped short of full internationalization – for example, system error messages are in Unixese, not Japanese – we believe Plan 9 is the first system to treat the representation of all major languages on a uniform, equal footing throughout all its software.

Introduction

The world is multilingual but most computer systems are based on English and ASCII. The release of Plan 9 [Pike90], a new distributed operating system from Bell Laboratories, seemed a good occasion to correct this chauvinism. It is easier to make such deep changes when building new systems than by refitting old ones.

The ANSI C standard [ANSIC] contains some guidance on the matter of 'wide' and 'multi-byte' characters but falls far short of solving the myriad associated problems. We could find no literature on how to convert a *system* to larger character sets, although some individual *programs* had been converted. This paper reports what we discovered as we explored the problem of representing multilingual text at all levels of an operating system, from the file system and kernel through the applications and up to the window system and display.

Plan 9 has not been 'internationalized': its manuals are in English, its error messages are in English, and it can display text that goes from left to right only. But before we can address these other problems, we need to handle, uniformly and comfortably, the textual representation of all the major written languages. That subproblem is richer than we had anticipated.

Standards

Our first step was to select a standard. At the time (January 1992), there were only two viable options: ISO 10646 [ISO10646] and Unicode [Unicode], with documents still in the draft stage.

ISO 10646 was not very attractive to us. The standard defines a sparse set of 32-bit characters, which would be hard to implement and have punitive storage requirements. Also, the standard attempts to mollify national interests by allocating 16-bit subspaces to national committees to partition individually. The suggested mode of use is to "flip" between separate national standards to implement the international standard. This did not strike us as a sound basis for a character set. As well, transmitting 32-bit values in a byte stream, such as in pipes, would be expensive and hard to implement. Since the standard does not define a byte order for such transmission, the byte stream would also have to carry state to enable the values to be recovered.

Unicode is a proposal by a consortium of mostly American computer companies formed to protest the technical failings of ISO 10646. Unicode defines a uniform 16-bit code based on the principle of unification: two characters are the same if they look the same even though they are from different languages. This principle, called Han unification, allows the large Japanese, Chinese, and Korean character sets to be packed comfortably into a 16-bit representation.

We chose Unicode for its technical merits and because its code space was better defined. Moreover, the existence of Unicode was derailing the ISO 10646 standard. ISO 10646 is now in its second draft and has only one 16-bit group defined, which is almost exactly Unicode. Most people expect the two standards bodies to reach a détente so that ISO 10646 and Unicode will represent the same character set.

Unicode defines an adequate character set but an unreasonable representation. The Unicode standard states that all characters are 16 bits wide and are communicated and stored in 16-bit units. It also reserves a pair of characters (hexadecimal FFFE and FEFF) to detect byte order in transmitted text, requiring state in the byte stream. (The Unicode committee was thinking of files, not pipes.) To adopt Unicode, we would have had to convert all text going into and out of Plan 9 between ASCII and Unicode, which cannot be done. Within a single program, in command of all its input and output, it is possible to define characters as 16-bit quantities; in the context of a networked system with hundreds of applications on diverse machines by different manufacturers, it is impossible.

We needed a way to adapt Unicode to the tools–and–pipes model of text processing embodied by the Unix system. To do that, we needed an ASCII–compatible textual representation of Unicode for transmission and storage. In the ISO standard there is an informative (non–required) Annex called UTF that provides a byte stream encoding of the 32–bit ISO code. The encoding uses multibyte sequences composed from the 190 printable characters of Latin–1 to represent character values larger than 159.

The UTF encoding has several good properties. By far the most important is that a byte in the ASCII range 0–127 represents itself in UTF. Thus UTF is backward compatible with ASCII.

UTF has other advantages. It is a byte encoding and is therefore byte–order independent. ASCII control characters appear in the byte stream only as themselves, never as an element of a sequence encoding another character, so newline bytes separate lines of UTF text. Finally, ANSI C's `strcmp` function applied to UTF strings preserves the ordering of Unicode characters.

To encode and decode UTF is expensive (involving multiplication, division, and modulo operations) but workable. UTF's major disadvantage is that the encoding is not self–synchronizing. It is in general impossible to find the character boundaries in a UTF string without reading from the beginning of the string, although in practice control characters such as newlines, tabs, and blanks provide synchronization points.

In August 1992, X–Open circulated a proposal for another UTF–like byte encoding of Unicode. Their major concern was that an embedded character in a file name (in particular a slash) could be part of an escape sequence in UTF and therefore confuse a traditional file system. Their proposal would allow all 7–bit ASCII characters to represent themselves *and only themselves* in text. Multibyte sequences would contain only characters with the high bit set. We proposed a modification to the new UTF that would address our synchronization problem. The modified new proposal is now informally called UTF–2 and is being proposed as another informative Annex to ISO 10646.

The model for text in Plan 9 is chosen from these three standards[1]: the Unicode character set encoded as byte stream by UTF–2, from an X–Open proposed modification of Annex F of ISO 10646. Although this may seem like a precarious position for us to adopt, it is not as bad as it sounds. If, as expected, ISO adopts Unicode as Group 0 of 10646 and ISO publishes UTF–2 as an Annex, then Plan 9 will be ISO/UTF–2 compatible.

[1] "That's the nice thing about standards – there's so many to choose from." – Andy Tannenbaum (no, the other one)

There are a couple of aspects of Unicode we have not faced. One is the issue of right–to–left text such as Hebrew or Arabic. Since that is an issue of display, not representation, we believe we can defer that problem for the moment without affecting our ability to solve it later. Another issue is diacriticals, which cause overstriking of multiple Unicode characters. Again, these are display issues and, since the Unicode committee is still deciding their finer points, we felt comfortable deferring. Mañana.

Although we converted Plan 9 in the altruistic interests of serving foreign languages, we have found the large character set attractive for other reasons. Unicode includes many characters – mathematical symbols, scientific notation, more general punctuation, and more – that we now use daily in our work. We no longer test our imaginations to find ways to include non–ASCII symbols in our text; why type a trigram like `:-)` when you can use the character ☺? Most compelling is the ability to absorb documents and data that contain non–ASCII characters; our browser for the Oxford English Dictionary lets us see the dictionary as it really is, with pronunciation in the IPA font, foreign phrases properly rendered, and so on, *in plain text.*

Throughout this paper, except when stated otherwise, the term 'UTF' refers to the UTF–2 encoding of Unicode characters as adopted by Plan 9.

C Compiler

The first program to be converted to UTF was the C Compiler. There are two levels of conversion. On the syntactic level, input to the C compiler is UTF; on the semantic level, the C language needs to define how compiled programs manipulate the UTF set.

The syntactic part is simple. The ANSI C language standard defines the source character set to be ASCII. Since UTF is backward compatible with ASCII, the compiler needs little change. The only places where a larger character set is allowed are in character constants, strings, and comments. Since 7–bit ASCII characters can represent only themselves in UTF, the compiler does not have to be careful while looking for the termination of a string or comment.

The Plan 9 compiler extends ANSI C to treat any Unicode character with a value outside of the ASCII range as an alphabetic. To a Greek programmer or an English mathematician, α is a sensible and now valid variable name.

On the semantic level, ANSI C allows, but does not tie down, the notion of a *wide character* and admits string and character constants of this type. We chose the wide character type to be `unsigned short`. In the libraries, the word `Rune` is defined by a `typedef` to be equivalent to `unsigned short` and is used to signify a Unicode character.

There are surprises; for example:

`L'x'`	is 120
`'x'`	is 120
`L'ÿ'`	is 55
`'ÿ'`	is –1, stdio `EOF` (if `char` is signed)
`L'α'`	is 945
`'α'`	is illegal

In the string constants,

```
"こんにちは 世界"
L"こんにちは 世界",
```

the former is an array of `char`s with 22 elements and a null byte, while the latter is an array of `unsigned short`s (`Rune`s) with 8 elements and a null `Rune`.

The Plan 9 library provides an output conversion function, `print` (analogous to `printf`), with formats `%c`, `%C`, `%s`, and `%S`. Since `print` produces text, its output is always UTF. The character conversion `%c` (lower case) masks its argument to 8 bits before converting to UTF. Thus `L'ÿ'` and `'ÿ'` printed under `%c` will be identical, but `L'α'` will print as the Unicode character with decimal value 177. The character conversion `%C` (upper case) masks its argument to 16 bits before converting to UTF. Thus `L'ÿ'` and `L'α'` will print correctly under `%C`, but `'ÿ'` will not. The conversion `%s` (lower case) expects a pointer to `char` and copies UTF sequences up to a null byte. The conversion `%S` (upper case) expects a pointer to `Rune` and performs sequential `%C` conversions until a null `Rune` is encountered.

Another problem in format conversion is the definition of `%10s`: does the number refer to bytes or characters? We decided that such formats were most often used to align output columns and so made the number count characters. Some programs, however, use the count to place blank-padded strings in fixed-sized arrays. These programs must be found and corrected.

Here is a complete example:

```
#include <u.h>

char c[] = "こんにちは 世界";
Rune s[] = L"こんにちは 世界";

main(void)
{
    print("%d, %d\n", sizeof(c),
            sizeof(s));
    print("%s\n", c);
    print("%S\n", s);
}
```

This program prints `23, 18` and then two identical lines of UTF text. In practice, `%S` and `L"..."` are rare in programs; one reason is that most formatted I/O is done in unconverted UTF.

Ramifications

All programs in Plan 9 now read and write text as UTF, not ASCII. This change breaks two deep-rooted symmetries implicit in most C programs:

1. A character is no longer a `char`.
2. The internal representation (Unicode) of a character now differs from its external representation (UTF).

In the sections that follow, we show how these issues were faced in the layers of system software from the operating system up to the applications. The effects are wide-reaching and often surprising.

Operating system

Since UTF is the only format for text in Plan 9, the interface to the operating system had to be converted to UTF. Text strings cross the interface in several places: command arguments, file names, user names (people can log in using their native name), error messages, and miscellaneous minor places such as commands to the I/O system. Little change was required: null-terminated UTF strings are equivalent to null-terminated ASCII strings for most purposes of the operating system. The library routines described in the next section made that change straightforward.

The window system, once called `8.5`, is now rightfully called `8½`.

Libraries

A header file included by all programs (see [Pike92]) declares the `Rune` type to hold 16-bit character values:

```
typedef unsigned short Rune;
```

Also defined are several constants relevant to UTF:

```
enum
{
  UTFmax    = 3,
      /* maximum bytes per rune */
  Runesync  = 0x80,
     /* cannot represent part of
        a UTF sequence (<) */
  Runeself  = 0x80,
     /* rune and UTF sequences
        are the same (<) */
  Runeerror = 0x80,
     /* decoding error in UTF */
};
```

(With the original UTF, `Runesync` was hexadecimal 21 and `Runeself` was A0.) `UTFmax` bytes are sufficient to hold the UTF encoding of any Unicode character. Characters of value less than `Runesync` only appear in a UTF string as themselves, never as part of a sequence encoding another character. Characters of value less than `Runeself` encode into single bytes of the same value. Finally, when the library detects errors in UTF input – byte sequences that are not valid UTF sequences – it converts the first byte of the error

sequence to the character Runeerror. There is little a rune-oriented program can do when given bad data except exit, which is unreasonable, or carry on. Originally the conversion routines, described below, returned errors when given invalid UTF, but we found ourselves repeatedly checking for errors and ignoring them. We therefore decided to convert a bad sequence to a valid rune and continue processing. (The ANSI C routines, on the other hand, return errors.)

This technique does have the unfortunate property that converting invalid UTF byte strings in and out of runes does not preserve the input, but this circumstance only occurs when non-textual input is given to a textual program. Unicode defines an error character, value FFFD, to represent characters from other sets that are not represented in Unicode. The Runeerror character is a different concept, related to UTF rather than Unicode, so we chose a different character for it.

The Plan 9 C library contains a number of routines for manipulating runes. The first set converts between runes and UTF strings:

```
extern int runetochar(char*, Rune*);
extern int chartorune(Rune*, char*);
extern int runelen(long);
extern int fullrune(char*, int);
```

Runetochar translates a single Rune to a UTF sequence and returns the number of bytes produced. Chartorune goes the other way, reporting how many bytes were consumed. Runelen returns the number of bytes in the UTF encoding of a rune. Fullrune examines a UTF string up to a specified number of bytes and reports whether the string begins with a complete UTF encoding. All these routines use the Runeerror character to work around encoding problems.

There is also a set of routines for examining null-terminated UTF strings, based on the model of the ANSI standard str routines, but with utf substituted for str and rune for chr:

```
extern int   utflen(char*);
extern char* utfrune(char*, long);
extern char* utfrrune(char*, long);
extern char* utfutf(char*, char*);
```

Utflen returns the number of runes in a UTF string; utfrune returns a pointer to the first occurrence of a rune in a UTF string; and utfrrune a pointer to the last. Utfutf searches for the first occurrence of a UTF string in another UTF string. Given the synchronizing property of UTF-2, utfutf is the same as strstr if the arguments point to valid UTF strings.

It is a mistake to use strchr or strrchr unless searching for a 7-bit ASCII character, that is, a character less than Runeself.

We have no routines for manipulating null-terminated arrays of Runes. Although they should probably exist for completeness, we have found no need for them, for the same reason that %S and L"..." are rarely used.

Most Plan 9 programs use a new buffered I/O library, BIO, in place of Standard I/O. BIO contains routines to read and write UTF streams, converting to and from runes. Bgetrune returns, as a Rune within a long, the next character in the UTF input stream; Bputrune takes a rune and writes its UTF representation. Bungetrune puts a rune back into the input stream for rereading.

Plan 9 programs use a simple set of macros to process command line arguments. Converting these macros to UTF automatically updated the argument processing of most programs. In general, argument flag names can no longer be held in bytes and arrays of 256 bytes cannot be used to hold a set of flags.

We have done nothing analogous to ANSI C's locales, partly because we do not feel qualified to define locales and partly because we remain unconvinced of that model for dealing with the problems. That is really more an issue of internationalization than conversion to a larger character set; on the other hand, because we have chosen a single character set that encompasses most languages, some of the need for locales is eliminated. (We have a utility, tcs, that translates between UTF and other character sets.)

There are several reasons why our library does not follow the ANSI design for wide and multi-byte characters. The ANSI model was designed by a committee, untried, almost as an afterthought, whereas we wanted to design as we built. (We made several major changes to the interface as we became familiar with the problems involved.) We disagree with ANSI C's handling of invalid multi-byte sequences. Also, the ANSI C library is incomplete: although it contains some crucial routines for handling wide and multi-byte characters, there are some serious omissions. For example, our software can exploit the fact that UTF preserves ASCII characters in the byte stream. We could remove that assumption by replacing all calls to strchr with utfrune and so on. (Because of the weaker properties of the original UTF, we have actually done so.) ANSI C cannot: the standard says nothing about the representation, so portable code should *never* call strchr, yet there is no ANSI equivalent to utfrune. ANSI C simultaneously invalidates strchr and offers no replacement.

Finally, ANSI did nothing to integrate wide characters into the I/O system: it gives no method for printing wide characters. We therefore needed to invent some things and decided to invent everything. In the end, some of our entry points do correspond closely to ANSI routines – for example, chartorune and runetochar are similar to mbtowc and wctomb – but Plan 9's library defines more functionality, enough to write real applications comfortably.

Converting the tools

The source for our tools and applications had already been converted to work with Latin–1, so it was '8–bit safe', but the conversion to Unicode and UTF is more involved. Some programs needed no change at all: `cat`, for instance, interprets its argument strings, delivered in UTF, as file names that it passes uninterpreted to the `open` system call, and then just copies bytes from its input to its output; it never makes decisions based on the values of the bytes. (Plan 9 `cat` has no options such as `-v` to complicate matters.) Most programs, however, needed modest change.

It is difficult to find automatically the places that need attention, but `grep` helps. Software that uses the libraries conscientiously can be searched for calls to library routines that examine bytes as characters: `strchr`, `strrchr`, `strstr`, etc. Replacing these by calls to `utfrune`, `utfrrune`, and `utfutf` is enough to fix many programs. Few tools actually need to operate on runes internally; more typically they need only to look for the final slash in a file name and similar trivial tasks. Of the 170 C source programs in the top levels of `/sys/src/cmd`, only 23 now contain the word `Rune`.

The programs that *do* store runes internally are mostly those whose *raison d'être* is character manipulation: `sam` (the text editor), `sed`, `sort`, `tr`, `troff`, `8½` (the window system and terminal emulator), and so on. To decide whether to compute using runes or UTF–encoded byte strings requires balancing the cost of converting the data when read and written against the cost of converting relevant text on demand. For programs such as editors that run a long time with a relatively constant dataset, runes are the better choice. There are space considerations too, but they are more complicated: plain ASCII text grows when converted to runes; UTF–encoded Japanese shrinks.

Again, it is hard to automate the conversion of a program from `chars` to `Runes`. It is not enough just to change the type of variables; the assumption that bytes and characters are equivalent can be insidious. For instance, to clear a character array by

```
memset(buf, 0, BUFSIZE)
```

becomes wrong if `buf` is changed from an array of `chars` to an array of `Runes`. Any program that indexes tables based on character values needs rethinking. Consider `tr`, which originally used multiple 256–byte arrays for the mapping. The naïve conversion would yield multiple 65536–rune arrays. Instead Plan 9 `tr` saves space by building in effect a run–encoded version of the map.

`Sort` has related problems. The cooperation of UTF and `strcmp` means that a simple sort – one with no options – can be done on the original UTF strings using `strcmp`. With sorting options enabled, however, `sort` may need to convert its input to runes: for example, option `-tα` requires searching for alphas in the input text to crack the input into fields. The field specifier `+3.2` refers to 2 runes beyond the third field. Some of the other options are hopelessly provincial: consider the case–folding and dictionary order options (Japanese doesn't even have an official dictionary order) or `-M` which compares by case–insensitive English month name. Handling these options involves the larger issues of internationalization and is beyond the scope of this paper and our expertise. Plan 9 `sort` works sensibly with options that make sense relative to the input. The simple and most important options are, however, usually meaningful. In particular, `sort` sorts UTF into the same order that `look` expects.

Regular expression–matching algorithms need rethinking to be applied to UTF text. Deterministic automata are usually applied to bytes; converting them to operate on variable–sized byte sequences is awkward. On the other hand, converting the input stream to runes adds measurable expense and the state tables expand from size 256 to 65536; it can be expensive just to generate them. For simple string searching, the Boyer–Moore algorithm works with UTF provided the input is guaranteed to be only valid UTF strings; however, it does not work with the old UTF encoding. At a more mundane level, even character classes are harder: the usual bit–vector representation within a non–deterministic automaton is unwieldy with 65536 characters in the alphabet.

We compromised. An existing library for compiling and executing regular expressions was adapted to work on runes, with two entry points for searching in arrays of runes and arrays of chars (the pattern is always UTF text). Character classes are represented internally as runs of runes; the reserved Unicode value `FFFF` marks the end of the class. Then *all* utilities that use regular expressions – editors, `grep`, `awk`, etc. – except the shell, whose notation was grandfathered, were converted to use the library. For some programs, there was a concomitant loss of performance, but there was also a strong advantage. To our knowledge, Plan 9 is the only Unix–like system that has a single definition and implementation of regular expressions; patterns are written and interpreted identically by all the programs in the system.

A handful of programs have the notion of character built into them so strongly as to confuse the issue of what they should do with UTF input. Such programs were treated as individual special cases. For example, `wc` is, by default, unchanged in behavior and output; a new option, `-r`, counts the number of correctly encoded runes – valid UTF sequences – in its input; `-b` the number of invalid sequences.

It took us several months to convert all the software in the system to Unicode and the old UTF. When we decided to convert from that to the new UTF, only three things needed to be done. First, we

rewrote the library routines to encode and decode the new UTF. This took an evening. Next, we converted all the files containing UTF to the new encoding. We wrote a trivial program to look for non-ASCII bytes in text files and used a Plan 9 program called `tcs` (translate character set) to change encodings. Finally, we recompiled all the system software; the library interface was unchanged, so recompilation was sufficient to effect the transformation. The second two steps were done concurrently and took an afternoon. We concluded that the actual encoding is relatively unimportant to the software; the adoption of large characters and a byte-stream encoding *per se* are much deeper issues.

Graphics and fonts

Plan 9 provides only minimal support for plain text terminals. It is instead designed to be used with all character input and output mediated by a window system such as 8½. The window system and related software are responsible for the display of UTF text as Unicode character images. For plain text, the window system must provide a user-settable *font* that provides a (possibly empty) picture for each Unicode character. Fancier applications that use bold and Italic characters need multiple fonts storing multiple pictures for each Unicode value. All the issues are apparent, though, in just the problem of displaying a single image for each character, that is, the Unicode equivalent of a plain text terminal. With 128 or even 256 characters, a font can be just an array of bitmaps. With 65536 characters, a more sophisticated design is necessary. To store the ideographs for just Japanese as 16×16×1 bit images, the smallest they can reasonably be, takes over a quarter of a megabyte. Make the images a little larger, store more bits per pixel, and hold a copy in every running application, and the memory cost becomes unreasonable.

The structure of the bitmap graphics services is described at length elsewhere [Pike91]. In summary, the memory holding the bitmaps is stored in the same machine that has the display, mouse, and keyboard: the terminal in Plan 9 terminology, the workstation in others'. Access to that memory and associated services is provided by device files served by system software on the terminal. One of those files, `/dev/bitblt`, interprets messages written upon it as requests for actions corresponding to entry points in the graphics library: allocate a bitmap, execute a raster operation, draw a text string, etc. The window system acts as a multiplexer that mediates access to the services and resources of the terminal by simulating in each client window a set of files mirroring those provided by the system. That is, each window has a distinct `/dev/mouse`, `/dev/bitblt`, and so on through which applications drive graphical input and output.

One of the resources managed by 8½ and the terminal is the set of active *subfonts*. Each subfont holds the bitmaps and associated data structures for a sequential set of Unicode characters. Subfonts are stored in files and loaded into the terminal by 8½ or an application. For example, one subfont might hold the images of the first 256 characters of the Unicode space, corresponding to the Latin-1 character set; another might hold the standard phonetic character set, Unicode characters with value 0250 to 02A8. These files are collected in directories corresponding to various typefaces: `/lib/font/bit/pelm` contains the Pellucida Monospace character set, with subfonts holding the Latin-1, Greek, Cyrillic and other components of the typeface. A suffix on subfont files encodes (in a subfont-specific way) the size of the images: `/lib/font/bit/pelm/latin1.9` contains the Latin-1 Pellucida Monospace characters with lower case letters nine pixels high; the file `/lib/font/bit/jis/jis5400.16` contains 16-pixel high ideographs starting at Unicode value 5400.

The subfonts do not identify which portion of the Unicode space they cover. Instead, a font file, in plain text, describes how to assemble subfonts into a complete character set. The font file is presented as an argument to the window system to determine how plain text is displayed in text windows and applications. Here is the beginning of the font file `/lib/font/bit/pelm/jis.9.font`, which describes the layout of a font covering that portion of Unicode for which we have characters of typical display size, using Japanese characters to cover the Han space:

```
18      14
0x0000  0x00FF  latin1.9
0x0100  0x017E  latineur.9
0x0250  0x02E9  ipa.9
0x0386  0x03F5  greek.9
0x0400  0x0475  cyrillic.9
0x2000  0x2044  ../misc/genpunc.9
0x2070  0x208E  supsub.9
0x20A0  0x20AA  currency.9
0x2100  0x2138  ../misc/letterlike.9
0x2190  0x21EA  ../misc/arrows
0x2200  0x227F  ../misc/math1
0x2280  0x22F1  ../misc/math2
0x2300  0x232C  ../misc/tech
0x2500  0x257F  ../misc/chart
0x2600  0x266F  ../misc/ding
0x3000  0x303f  ../jis/jis3000.16
0x30a1  0x30fe  ../jis/katakana.16
0x3041  0x309e  ../jis/hiragana.16
0x4e00  0x4fff  ../jis/jis4e00.16
0x5000  0x51ff  ../jis/jis5000.16
...
```

The first two numbers set the interline spacing of the font (18 pixels) and the distance from the baseline to the top of the line (14 pixels). When characters are displayed, they are placed so as best to fit within those

constraints; characters too large to fit will be truncated. The rest of the file associates subfont files with portions of Unicode space. The first four such files are in the Pellucida Monospace typeface and directory; others reside in other directories. The file names are relative to the font file's own location.

There are several advantages to this two-level structure. First, it simultaneously breaks the huge Unicode space into manageable components and provides a unifying architecture for assembling fonts from disjoint pieces. Second, the structure promotes sharing. For example, we have only one set of Japanese characters but dozens of typefaces for the Latin-1 characters, and this structure permits us to store only one copy of the Japanese set but use it with any Roman typeface. Also, customization is easy. English-speaking users who don't need Japanese characters but may want to read an on-line Oxford English Dictionary can assemble a custom font with the Latin-1 (or even just ASCII) characters and the International Phonetic Alphabet (IPA). Moreover, to do so requires just editing a plain text file, not using a special font editing tool. Finally, the structure guides the design of caching protocols to improve performance and memory usage.

To load a complete Unicode character set into each application would consume too much memory and, particularly on slow terminal lines, would take unreasonably long. Instead, Plan 9 assembles a multi-level cache structure for each font. An application opens a font file, reads and parses it, and allocates a data structure. A message written to `/dev/bitblt` allocates an associated structure held in the terminal, in particular, a bitmap to act as a cache for recently used character images. Other messages copy these images to bitmaps such as the screen by loading characters from subfonts into the cache on demand and from there to the destination bitmap. The protocol to draw characters is in terms of cache indices, not Unicode character number or UTF sequences. These details are hidden from the application, which instead sees only a subroutine to draw a string in a bitmap from a given font, functions to discover character size information, and routines to allocate and to free fonts.

As needed, whole subfonts are opened by the graphics library, read, and then downloaded to the terminal. They are held open by the library in an LRU-replacement list. Even when the program closes a subfont, it is retained in the terminal for later use. When the application opens the subfont, it asks the terminal if it already has a copy to avoid reading it from the file server if possible. This level of cache has the property that the bitmaps for, say, all the Japanese characters are stored only once, in the terminal; the applications read only size and width information from the terminal and share the images.

The sizes of the character and subfont caches held by the application are adaptive. A simple algorithm monitors the cache miss rate to enlarge and shrink the caches as required. The size of the character cache is limited to 2048 images maximum, which in practice seems enough even for Japanese text. For plain ASCII-like text it naturally stays around 128 images.

This mechanism sounds complicated but is implemented by only about 500 lines in the library and considerably less in each of the terminal's graphics driver and 8½. It has the advantage that only characters that are being used are loaded into memory. It is also efficient: if the characters being drawn are in the cache the extra overhead is negligible. It works particularly well for alphabetic character sets, but also adapts on demand for ideographic sets. When a user first looks at Japanese text, it takes a few seconds to read all the font data, but thereafter the text is drawn almost as fast as regular text (the images are larger, so draw a little slower). Also, because the bitmaps are remembered by the terminal, if a second application then looks at Japanese text it starts faster than the first.

We considered building a 'font server' to cache character images and associated data for the applications, the window system, and the terminal. We rejected this design because, although isolating many of the problems of font management into a separate program, it didn't simplify the applications. Moreover, in a distributed system such as Plan 9 it is easy to have too many special purpose servers. Making the management of the fonts the concern of only the essential components simplifies the system and makes bootstrapping less intricate.

Input

A completely different problem is how to type Unicode characters as input to the system. We selected an unused key on our ASCII keyboards to serve as a prefix for multi-keystroke sequences that generate Unicode characters. For example, the character ü is generated by the prefix key (typically ALT or Compose) followed by a double quote and a lower-case u. When that character is read by the application, from the file `/dev/cons`, it is of course presented as its UTF encoding. Such sequences generate characters from an arbitrary set that includes all of Latin-1 plus a selection of mathematical and technical characters. An arbitrary Unicode character may be generated by typing the prefix, an upper case X, and four hexadecimal digits that identify the Unicode value.

These simple mechanisms are adequate for most of our day-to-day needs: it's easy to remember to type 'ALT 1 2' for ½ or 'ALT accent letter' for accented Latin letters. For the occasional unusual character, the cut and paste features of 8½ serve well. A program called (perhaps misleadingly) `unicode` takes as argument a hexadecimal value, and prints the

UTF representation of that character, which may then be picked up with the mouse and used as input.

These methods are clearly unsatisfactory when working in a non–English language. In the native country of such a language the appropriate keyboard is likely to be at hand. But it's also reasonable – especially now the system handles Unicode – to work in a language foreign to the keyboard.

For alphabetic languages such as Greek or Russian, it is straightforward to construct a program that does phonetic substitution, so that, for example, typing a Latin 'a' yields the Greek 'α'. Within Plan 9, such a program can be inserted transparently between the real keyboard and a program such as the window system, providing a manageable input device for such languages.

For ideographic languages such as Chinese or Japanese the problem is harder. Native users of such languages have adopted methods for dealing with Latin keyboards that involve a hybrid technique based on phonetics to generate a list of possible symbols followed by menu selection to choose the desired one. Such methods can be effective, but their design must be rooted in information about the language unknown to non–native speakers. (`Cxterm`, a Chinese terminal emulator built by and for Chinese programmers, employs such a technique [Pong and Zhang].) Although the technical problem of implementing such a device is easy in Plan 9 – it is just an elaboration of the technique for alphabetic languages – our lack of familiarity with such languages has restrained our enthusiasm for building one.

The input problem is technically the least interesting but perhaps emotionally the most important of the problems of converting a system to an international character set. Beyond that remain the deeper problems of internationalization such as multi–lingual error messages and command names, problems we are not qualified to solve. With the ability to treat text of most languages on an equal footing, though, we can begin down that path. Perhaps people in non–English speaking countries will consider adopting Plan 9, solving the input problem locally – perhaps just by plugging in their local terminals – and begin to use a system with at least the capacity to be international.

Acknowledgements

Dennis Ritchie provided consultation and encouragement. Bob Flandrena converted most of the standard tools to UTF. Brian Kernighan suffered cheerfully with several inadequate implementations and converted `troff` to UTF. Rich Drechsler converted his Postscript driver to UTF. John Hobby built the Postscript ☺. We thank them all.

Author Information

Rob Pike, well known for his appearances on "Late Night with David Letterman", is also a Member of Technical Staff at AT&T Bell Laboratories in Murray Hill, New Jersey, where he has been since 1980, the same year he won the Olympic silver medal in Archery. In 1981 he wrote the first bitmap window system for Unix systems, and has since written nine more. With Bart Locanthi he designed the Blit terminal; with Brian Kernighan he wrote *The Unix Programming Environment.* A shuttle mission nearly launched a gamma–ray telescope he designed. He is a Canadian citizen and has never written a program that uses cursor addressing.

Ken Thompson was born in New Orleans, Louisiana in 1943. He attended the University of California at Berkeley and received B.S. and M.S. degrees in Electrical Engineering. In 1966 he joined the Computing Science Research Center of Bell Laboratories where he has worked until the present. He was involved in Bell Laboratories' participation in the Multics project. Mr. Thompson is one of the principal designers of the UNIX time–sharing system. He is also one of the principal designers of the former World Computer Chess Champion, "Belle".

References

[ANSIC] *American National Standard for Information Systems – Programming Language C*, American National Standards Institute, Inc., New York, 1990.

[ISO10646] ISO/IEC DIS 10646–1 *Information technology – Universal Multiple–Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plane.*

[Pike90] R. Pike, D. Presotto, K. Thompson, H. Trickey, "Plan 9 from Bell Labs", UKUUG Proc. of the Summer 1990 Conf., London, England, 1990.

[Pike91] Pike, R., "8.5, The Plan 9 Window System", USENIX Summer Conf. Proc., Nashville, 1991.

[Pike92] Pike, R., "How to Use the Plan 9 C Compiler", in *The Plan 9 Programmer's Manual*, AT&T Bell Laboratories, Murray Hill, NJ, 1992.

[Pong and Zhang] Man–Chi Pong and Yongguang Zhang, "cxterm: A Chinese Terminal Emulator for the X Window System", *Software–Practice and Experience*, Vol 22(1), 809–926, October 1992.

[Unicode] *The Unicode Standard, Worldwide Character Encoding, Version 1.0, Volume 1*, The Unicode Consortium, Addison Wesley, New York, 1991.

Es: A shell with higher-order functions

Paul Haahr – Adobe Systems Incorporated
Byron Rakitzis – Network Appliance Corporation

ABSTRACT

In the fall of 1990, one of us (Rakitzis) re-implemented the Plan 9 command interpreter, *rc*, for use as a UNIX shell. Experience with that shell led us to wonder whether a more general approach to the design of shells was possible, and this paper describes the result of that experimentation. We applied concepts from modern functional programming languages, such as Scheme and ML, to shells, which typically are more concerned with UNIX features than language design. Our shell is both simple and highly programmable. By exposing many of the internals and adopting constructs from functional programming languages, we have created a shell which supports new paradigms for programmers.

Although most users think of the shell as an interactive command interpreter, it is really a programming language in which each statement runs a command. Because it must satisfy both the interactive and programming aspects of command execution, it is a strange language, shaped as much by history as by design.

— Brian Kernighan & Rob Pike [1]

Introduction

A shell is both a programming language and the core of an interactive environment. The ancestor of most current shells is the 7th Edition Bourne shell[2], which is characterized by simple semantics, a minimal set of interactive features, and syntax that is all too reminiscent of Algol. One recent shell, *rc*[3], substituted a cleaner syntax but kept most of the Bourne shell's attributes. However, most recent developments in shells (e.g., *csh*, *ksh*, *zsh*) have focused on improving the interactive environment without changing the structure of the underlying language – shells have proven to be resistant to innovation in programming languages.

While *rc* was an experiment in adding modern syntax to Bourne shell semantics, *es* is an exploration of new semantics combined with *rc*-influenced syntax: *es* has lexically scoped variables, first-class functions, and an exception mechanism, which are concepts borrowed from modern programming languages such as Scheme and ML[4, 5].

In *es*, almost all standard shell constructs (e.g., pipes and redirection) are translated into a uniform representation: function calls. The primitive functions which implement those constructs can be manipulated the same way as all other functions: invoked, replaced, or passed as arguments to other functions. The ability to replace primitive functions in *es* is key to its extensibility; for example, a user can override the definition of pipes to cause remote execution, or the path-searching machinery to implement a path look-up cache.

At a superficial level, *es* looks like most UNIX shells. The syntax for pipes, redirection, background jobs, etc., is unchanged from the Bourne shell. *Es*'s programming constructs are new, but reminiscent of *rc* and Tcl[6].

Es is freely redistributable, and is available by anonymous ftp from *ftp.white.toronto.edu*.

Using *es*

Commands

For simple commands, *es* resembles other shells. For example, newline usually acts as a command terminator. These are familiar commands which all work in *es*:

```
cd /tmp
rm Ex*
ps aux | grep '^byron' |
  awk '{print $2}' | xargs kill -9
```

For simple uses, *es* bears a close resemblance to *rc*. For this reason, the reader is referred to the paper on *rc* for a discussion of quoting rules, redirection, and so on. (The examples shown here, however, will try to aim for a lowest common denominator of shell syntax, so that an understanding of *rc* is not a prerequisite for understanding this paper.)

Functions

Es can be programmed through the use of shell functions. Here is a simple function to print the date in *yy-mm-dd* format:

```
fn d {
    date +%y-%m-%d
}
```

Functions can also be called with arguments. *Es* allows parameters to be specified to functions by placing them between the function name and the open-brace. This function takes a command `cmd` and arguments `args` and applies the command to each argument in turn:

```
fn apply cmd args {
    for (i = $args)
        $cmd $i
}
```

For example:[1]

```
es> apply echo testing 1.. 2.. 3..
testing
1..
2..
3..
```

Note that `apply` was called with more than two arguments; *es* assigns arguments to parameters one-to-one, and any leftovers are assigned to the last parameter. For example:

```
es> fn rev3 a b c {
    echo $c $b $a
}
es> rev3 1 2 3 4 5
3 4 5 2 1
```

If there are fewer arguments than parameters, *es* leaves the leftover parameters null:

```
es> rev3 1
1
```

So far we have only seen simple strings passed as arguments. However, *es* functions can also take program fragments (enclosed in braces) as arguments. For example, the `apply` function defined above can be used with program fragments typed directly on the command line:

```
es> apply @ i {cd $i; rm -f *} \
                        /tmp /usr/tmp
```

This command contains a lot to understand, so let us break it up slowly.

In any other shell, this command would usually be split up into two separate commands:

```
es> fn cd-rm i {
    cd $i
    rm -f *
}
es> apply cd-rm /tmp /usr/tmp
```

Therefore, the construct

```
@ i {cd $i; rm -f *}
```

is just a way of inlining a function on the command-line. This is called a *lambda*.[2] It takes the form

`@` *parameters* `{` *commands* `}`

In effect, a lambda is a procedure "waiting to happen." For example, it is possible to type:

```
es> @ i {cd $i; rm -f *} /tmp
```

directly at the shell, and this runs the inlined function directly on the argument `/tmp`.

There is one more thing to notice: the inline function that was supplied to `apply` had a parameter named `i`, and the `apply` function itself used a reference to a variable called `i`. Note that the two uses did not conflict: that is because *es* function parameters are *lexically scoped*, much as variables are in C and Scheme.

Variables

The similarity between shell functions and lambdas is not accidental. In fact, function definitions are rewritten as assignments of lambdas to shell variables. Thus these two *es* commands are entirely equivalent:

```
fn echon args {echo -n $args}
fn-echon = @ args {echo -n $args}
```

In order not to conflict with regular variables, function variables have the prefix `fn-` prepended to their names. This mechanism is also used at execution time; when a name like `apply` is seen by *es*, it first looks in its symbol table for a variable by the name `fn-apply`. Of course, it is always possible to execute the contents of any variable by dereferencing it explicitly with a dollar sign:

```
es> silly-command = {echo hi}
es> $silly-command
hi
```

The previous examples also show that variables can be set to contain program fragments as well as simple strings. In fact, the two can be intermixed:

```
es> mixed = {ls} hello, {wc} world
es> echo $mixed(2) $mixed(4)
hello, world
es> $mixed(1) | $mixed(3)
      61      61     478
```

Variables can hold a list of commands, or even a list of lambdas. This makes variables into versatile tools. For example, a variable could be used as a function dispatch table.

[1] In our examples, we use "`es>` " as *es*'s prompt. The default prompt, which may be overridden, is "`; `" which is interpreted by *es* as a null command followed by a command separator. Thus, whole lines, including prompts, can be cut and pasted back to the shell for re-execution. In examples, an italic fixed width font indicates user input.

[2] The keyword `@` introduces the lambda. Since `@` is not a special character in *es* it must be surrounded by white space. `@` is a poor substitute for the letter λ, but it was one of the few characters left on a standard keyboard which did not already have a special meaning.

Binding

In the section on functions, we mentioned that function parameters are lexically scoped. It is also possible to use lexically-scoped variables directly. For example, in order to avoid interfering with a global instance of `i`, the following scoping syntax can be used:

```
let (var = value) {
    commands which use $var
}
```

Lexical binding is useful in shell functions, where it becomes important to have shell functions that do not clobber each others' variables.

Es code fragments, whether used as arguments to commands or stored in variables, capture the values of enclosing lexically scoped values. For example,

```
es> let (h=hello; w=world) {
    hi = { echo $h, $w }
}
es> $hi
hello, world
```

One use of lexical binding is in redefining functions. A new definition can store the previous definition in a lexically scoped variable, so that it is only available to the new function. This feature can be used to define a function for tracing calls to other functions:

```
fn trace functions {
    for (func = $functions)
        let (old = $(fn-$func))
            fn $func args {
              echo calling $func $args
              $old $args
            }
}
```

The `trace` function redefines all the functions which are named on its command line with a function that prints the function name and arguments and then calls the previous definition, which is captured in the lexically bound variable `old`. Consider a recursive function `echo-nl` which prints its arguments, one per line:

```
es> fn echo-nl head tail {
    if {!~ $#head 0} {
        echo $head
        echo-nl $tail
    }
}
es> echo-nl a b c
a
b
c
```

Applying `trace` to this function yields:

```
es> trace echo-nl
es> echo-nl a b c
calling echo-nl a b c
a
calling echo-nl b c
b
calling echo-nl c
c
calling echo-nl
```

The reader should note that

```
! cmd
```

is *es*'s "not" command, which inverts the sense of the return value of *cmd*, and

```
~ subject pattern
```

matches *subject* against *pattern* and returns true if the subject is the same as the pattern. (In fact, the matching is a bit more sophisticated, for the pattern may include wildcards.)

Shells like the Bourne shell and *rc* support a form of local assignment known as *dynamic binding*. The shell syntax for this is typically:

```
var=value command
```

That notation conflicts with *es*'s syntax for assignment (where zero or more words are assigned to a variable), so dynamic binding has the syntax:

```
local (var = value) {
    commands which use $var
}
```

The difference between the two forms of binding can be seen in an example:

```
es> x = foo
es> let (x = bar) {
    echo $x
    fn lexical { echo $x }
}
bar
es> lexical
bar
es> local (x = baz) {
    echo $x
    fn dynamic { echo $x }
}
baz
es> dynamic
foo
```

Settor Variables

In addition to the prefix (`fn-`) for function execution described earlier, *es* uses another prefix to search for *settor variables*. A settor variable `set-`*foo* is a variable which gets evaluated every time the variable *foo* changes value. A good example of settor variable use is the `watch` function:

```
fn watch vars {
   for (var = $vars) {
      set-$var = @ {
         echo old $var '=' $$var
         echo new $var '=' $*
         return $*
      }
   }
}
```

Watch establishes a settor function for each of its parameters; this settor prints the old and new values of the variable to be set, like this:

```
es> watch x
es> x=foo bar
old x =
new x = foo bar
es> x=fubar
old x = foo bar
new x = fubar
```

Return Values

UNIX programs exit with a single number between 0 and 255 reported as their statuses. *Es* supplants the notion of an exit status with "rich" return values. An *es* function can return not only a number, but any object: a string, a program fragment, a lambda, or a list which mixes such values.

The return value of a command is accessed by prepending the command with <>:

```
es> fn hello-world {
   return 'hello, world'
}
es> echo <>{hello-world}
hello, world
```

This example shows rich return values being used to implement hierarchical lists:

```
fn cons a d {
   return @ f { $f $a $d }
}
fn car p { $p @ a d { return $a } }
fn cdr p { $p @ a d { return $d } }
```

The first function, cons, returns a function which takes as its argument another function to run on the parameters a and d. car and cdr each invoke the kind of function returned by cons, supplying as the argument a function which returns the first or second parameter, respectively. For example:

```
es> echo <>{car <>{cdr <>{
   cons 1 <>{cons 2 <>{cons 3 nil}}
}}}
2
```

Exceptions

In addition to traditional control flow constructs – loops, conditionals, subroutines – *es* has an exception mechanism which is used for implementing non-structured control flow. The built-in function throw raises an exception, which typically consists of a string which names the exception and other arguments which are specific to the named exception type. For example, the exception error is caught by the default interpreter loop, which treats the remaining arguments as an error message. Thus:

```
es> fn in dir cmd {
   if {~ $#dir 0} {
     throw error 'usage: in dir cmd'
   }
   fork     # run in a subshell
   cd $dir
   $cmd
}
es> in
usage: in dir cmd
es> in /tmp ls
webster.socket      yacc.312
```

By providing a routine which catches error exceptions, a programmer can intercept internal shell errors before the message gets printed.

Exceptions are also used to implement the break and return control flow constructs, and to provide a way for user code to interact with UNIX signals. While six error types are known to the interpreter and have special meanings, any set of arguments can be passed to throw.

Exceptions are trapped with the built-in catch, which typically takes the form

```
catch @ e args { handler } { body }
```

Catch first executes *body*; if no exception is raised, catch simply returns, passing along *body*'s return value. On the other hand, if anything invoked by *body* throws an exception, *handler* is run, with e bound to the exception that caused the problem. For example, the last two lines of in above can be replaced with:

```
catch @ e msg {
   if {~ $e error} {
      echo >[1=2] in $dir: $msg
   } {
      throw $e $msg
   }
} {
   cd $dir
   $cmd
}
```

to better identify for a user where an error came from:

```
es> in /temp ls
in /temp: chdir /temp:
        No such file or directory
```

Spoofing

Es's versatile functions and variables are only half of the story; the other part is that *es*'s shell syntax is just a front for calls on built-in functions. For example:

```
ls > /tmp/foo
```

is internally rewritten as

```
%create 1 /tmp/foo {ls}
```

before it is evaluated. `%create` is the built-in function which opens `/tmp/foo` on file-descriptor 1 and runs `ls`.

The value of this rewriting is that the `%create` function (and that of just about any other shell service) can be *spoofed*, that is, overridden by the user: when a new `%create` function is defined, the default action of redirection is overridden.

Furthermore, `%create` is not really the built-in file redirection service. It is a hook to the *primitive* `$&create`, which itself cannot be overridden. That means that it is always possible to access the underlying shell service, even when its hook has been reassigned.

Keeping this in mind, here is a spoof of the redirection operator that we have been discussing. This spoof is simple: if the file to be created exists (determined by running `test -f`), then the command is not run, similar to the C-shell's "noclobber" option:

```
fn %create fd file cmd {
    if {test -f $file} {
        throw error $file exists
    } {
        $&create $fd $file $cmd
    }
}
```

In fact, most redefinitions do not refer to the `$&`-forms explicitly, but capture references to them with lexical scoping. Thus, the above redefinition would usually appear as

```
let (create = $fn-%create)
    fn %create fd file cmd {
        if {test -f $file} {
            throw error $file exists
        } {
            $create $fd $file $cmd
        }
    }
```

The latter form is preferable because it allows multiple redefinitions of a function; the former version would always throw away any previous redefinitions.

Overriding traditional shell built-ins is another common example of spoofing. For example, a `cd` operation which also places the current directory in the title-bar of the window (via the hypothetical command `title`) can be written as:

```
es> let (pipe = $fn-%pipe) {
    fn %pipe first out in rest {
        if {~ $#out 0} {
            time $first
        } {
            $pipe {time $first} $out $in {%pipe $rest}
        }
    }
}
es> cat paper9 | tr -cs a-zA-Z0-9 '\012' | sort | uniq -c | sort -nr | sed 6q
 213 the
 150 a
 120 to
 115 of
 109 is
  96 and
    2r   0.3u   0.2s   cat paper9
    2r   0.3u   0.2s   tr -cs a-zA-Z0-9 \012
    2r   0.5u   0.2s   sort
    2r   0.4u   0.2s   uniq -c
    3r   0.2u   0.1s   sed 6q
    3r   0.6u   0.2s   sort -nr
```

Figure 1: Timing pipeline elements

```
let (cd = $fn-%cd)
   fn cd {
      $cd $*
      title '{pwd}
   }
```

Spoofing can also be used for tasks which other shells cannot do; one example is timing each element of a pipeline by spoofing `%pipe`, along the lines of the pipeline profiler suggested by Jon Bentley[7]; see Figure 1.

Many shells provide some mechanism for caching the full pathnames of executables which are looked up in a user's `$PATH`. *Es* does not provide this functionality in the shell, but it can easily be added by any user who wants it. The function `%pathsearch` (see Figure 2) is invoked to look-up non-absolute file names which are used as commands.

One other piece of *es* which can be replaced is the interpreter loop. In fact, the default interpreter is written in *es* itself; see Figure 3.

A few details from this example need further explanation. The exception `retry` is intercepted by `catch` when an exception handler is running, and causes the body of the `catch` routine to be re-run.

```
let (search = $fn-%pathsearch) {
   fn %pathsearch prog {
      let (file = <>{$search $prog}) {
         if {~ $#file 1 && ~ $file /*} {
            path-cache = $path-cache $prog
            fn-$prog = $file
         }
         return $file
      }
   }
}
fn recache {
   for (i = $path-cache)
      fn-$i =
   path-cache =
}
```

Figure 2: Path caching

```
fn %interactive-loop {
   let (result = 0) {
      catch @ e msg {
         if {~ $e eof} {
            return $result
         } {~ $e error} {
            echo >[1=2] $msg
         } {
            echo >[1=2] uncaught exception: $e $msg
         }
         throw retry
      } {
         while {} {
            %prompt
            let (cmd = <>{%parse $prompt}) {
               result = <>{$cmd}
            }
         }
      }
   }
}
```

Figure 3: Default interactive loop

`%parse` prints its first argument to standard error, reads a command (potentially more than one line long) from the current source of command input, and throws the `eof` exception when the input source is exhausted. The hook `%prompt` is provided for the user to redefine, and by default does nothing.

Other spoofing functions which either have been suggested or are in active use include: a version of `cd` which asks the user whether to create a directory if it does not already exist; versions of redirection and program execution which try spelling correction if files are not found; a `%pipe` to run pipeline elements on (different) remote machines to obtain parallel execution; automatic loading of shell functions; and replacing the function which is used for tilde expansion to support alternate definitions of home directories. Moreover, for debugging purposes, one can use `trace` on hook functions.

Implementation

Es is implemented in about 8000 lines of C. Although we estimate that about 1000 lines are devoted to portability issues between different versions of UNIX, there are also a number of workarounds that *es* must use in order to blend with UNIX. The `path` variable is a good example.

The *es* convention for path searching involves looking through the list elements of a variable called `path`. This has the advantage that all the usual list operations can be applied equally to `path` as any other variable. However, UNIX programs expect the path to be a colon-separated list stored in `PATH`. Hence *es* must maintain a copy of each variable, with a change in one reflected as a change in the other.

Initialization

Much of *es*'s initialization is actually done by an *es* script, called `initial.es`, which is converted by a shell script to a C character string at compile time and stored internally. The script illustrates how the default actions for *es*'s parser is set up, as well as features such as the `path`/`PATH` aliasing mentioned above.

Much of the script consists of lines like:

```
fn-%and        = $&and
fn-%append     = $&append
fn-%background = $&background
```

which bind the shell services such as short-circuit-and, backgrounding, etc., to the `%`-prefixed hook variables.

There are also a set of assignments which bind the built-in shell functions to their hook variables:

```
fn-.       = $&dot
fn-break   = $&break
fn-catch   = $&catch
```

The difference with these is that they are given names invoked directly by the user; "`.`" is the Bourne-compatible command for "sourcing" a file.

Finally, some settor functions are defined to work around UNIX path searching (and other) conventions. For example,

```
set-path = @ {
    local (set-PATH = )
        PATH = <>{%flatten : $*}
    return $*
}
set-PATH = @ {
    local (set-path = )
        path = <>{%fsplit : $*}
    return $*
}
```

A note on implementation: these functions temporarily assign their opposite-case settor cousin to null before making the assignment to the opposite-case variable. This avoids infinite recursion between the two settor functions.

The Environment

UNIX shells typically maintain a table of variable definitions which is passed on to child processes when they are created. This table is loosely referred to as the environment or the environment variables. Although traditionally the environment has been used to pass values of variables only, the duality of functions and variables in *es* has made it possible to pass down function definitions to subshells. (While *rc* also offered this functionality, it was more of a kludge arising from the restriction that there was not a separate space for "environment functions.")

Having functions in the environment brings them into the same conceptual framework as variables – they follow identical rules for creation, deletion, presence in the environment, and so on. Additionally, functions in the environment are an optimization for file I/O and parsing time. Since nearly all shell state can now be encoded in the environment, it becomes superfluous for a new instance of *es*, such as one started by *xterm*(1), to run a configuration file. Hence shell startup becomes very quick.

As a consequence of this support for the environment, a fair amount of *es* must be devoted to "unparsing" function definitions so that they may be passed as environment strings. This is complicated a bit more because the lexical environment of a function definition must be preserved at unparsing. This is best illustrated by an example:

```
es> let (a=b) fn foo {echo $a}
```

which lexically binds `b` to the variable `a` for the scope of this function definition. Therefore, the external representation of this function must make this information explicit. It is encoded as:

```
es> whatis foo
%closure(a=b)@ * {echo $a}
```

(Note that for cultural compatibility with other shells, functions with no named parameters use "*" for binding arguments.)

Interactions With Unix

Unlike most traditional shells, which have feature sets dictated by the UNIX system call interface, *es* contains features which do not interact well with UNIX itself. For example, rich return values make sense from shell functions (which are run inside the shell itself) but cannot be returned from shell scripts or other external programs, because the *exit/wait* interface only supports passing small integers. This has forced us to build some things into the shell which otherwise could be external.

The exception mechanism has similar problems. When an exception is raised from a shell function, it propagates as expected; if raised from a subshell, it cannot be propagated as one would like it to be: instead, a message is printed on exit from the subshell and a false exit status is returned. We consider this unfortunate, but there seemed no reasonable way to tie exception propagation to any existing UNIX mechanism. In particular, the signal machinery is unsuited to the task. In fact, signals complicate the control flow in the shell enough, and cause enough special cases throughout the shell, so as to be more of a nuisance than a benefit.

One other unfortunate consequence of our shoehorning *es* onto UNIX systems is the interaction between lexically scoped variables, the environment, and subshells. Two functions, for example, may have been defined in the same lexical scope. If one of them modifies a lexically scoped variable, that change will affect the variable as seen by the other function. On the other hand, if the functions are run in a subshell, the connection between their lexical scopes is lost as a consequence of them being exported in separate environment strings. This does not turn out to be a significant problem, but it does not seem intuitive to a programmer with a background in functional languages.

One restriction on *es* that arose because it had to work in a traditional UNIX environment is that lists are not hierarchical; that is, lists may not contain lists as elements. In order to be able to pass lists to external programs with the same semantics as passing them to shell functions, we had to restrict lists to the same structure as *exec*-style argument vectors. Therefore all lists are flattened, as in *rc* and *csh*.

Garbage Collection

Since *es* incorporates a true lambda calculus, it includes the ability to create true recursive structures, that is, objects which include pointers to themselves, either directly or indirectly. While this feature can be useful for programmers, it has the unfortunate consequence of making memory management in *es* more complex than that found in other shells. Simple memory reclamation strategies such as arena style allocation [8] or reference counting are unfortunately inadequate; a full garbage collection system is required to plug all memory leaks.

Based on our experience with *rc*'s memory use, we decided that a copying garbage collector would be appropriate for *es*. The observations leading to this conclusion were: (1) between two separate commands little memory is preserved (it roughly corresponds to the storage for environment variables); (2) command execution can consume large amounts of memory for a short time, especially when loops are involved; and, (3) however much memory is used, the working set of the shell will typically be much smaller than the physical memory available. Thus, we picked a strategy where we traded relatively fast collection times for being somewhat wasteful in the amount of memory used in exchange. While a generational garbage collector might have made sense for the same reasons that we picked a copying collector, we decided to avoid the added complexity implied by switching to the generational model.

During normal execution of the shell, memory is acquired by incrementing a pointer through a preallocated block. When this block is exhausted, all live pointers from outside of garbage collector memory, the *rootset*, are examined, and any structure that they point to is copied to a new block. When the rootset has been scanned, all the freshly copied data is scanned similarly, and the process is repeated until all reachable data has been copied to the new block. At this point, the memory request which triggered the collection should be able to succeed. If not, a larger block is allocated and the collection is redone.

During some parts of the shell's execution – notably while the *yacc* parser driver is running – it is not possible to identify all of the rootset, so garbage collection is disabled. If an allocation request is made during this time for which there is not enough memory available in the arena, a new chunk of memory is grabbed so that allocation can continue.

Garbage collectors have developed a reputation for being hard to debug. The collection routines themselves typically are not the source of the difficulty. Even more sophisticated algorithms than the one found in *es* are usually only a few hundred lines of code. Rather, the most common form of GC bug is failing to identify all elements of the rootset, since this is a rather open-ended problem which has implications for almost every routine. To find this form of bug, we used a modified version of the garbage collector which has two key features: (1) a collection is initiated at every allocation when the collector is not disabled, and (2) after a collection finishes, access to all the memory from the old

region is disabled.[3] Thus, any reference to a pointer in garbage collector space which could be invalidated by a collection immediately causes a memory protection fault. We strongly recommend this technique to anyone implementing a copying garbage collector.

There are two performance implications of the garbage collector; the first is that, occasionally, while the shell is running, all action must stop while the collector is invoked. This takes roughly 4% of the running time of the shell. More serious is that at the time of any potential allocation, either the collector must be disabled, or all pointers to structures in garbage collector memory must be identified, effectively requiring them to be in memory at known addresses, which defeats the registerization optimizations required for good performance from modern architectures. It is hard to quantify the performance consequences of this restriction.

The garbage collector consists of about 250 lines of code for the collector itself (plus another 300 lines of debugging code), along with numerous declarations that identify variables as being part of the rootset and small (typically 5 line) procedures to allocate, copy, and scan all the structure types allocated from collector space.

Future Work

There are several places in *es* where one would expect to be able to redefine the built-in behavior and no such hook exists. The most notable of these is the wildcard expansion, which behaves identically to that in traditional shells. We hope to expose some of the remaining pieces of *es* in future versions.

One of the least satisfying pieces of *es* is its parser. We have talked of the distinction between the core language and the full language; in fact, the translation of *syntactic sugar* (i.e., the convenient UNIX shell syntax presented to the user) to core language features is done in the same *yacc*-generated parser as the recognition of the core language. Unfortunately, this ties the full language in to the core very tightly, and offers little room for a user to extend the syntax of the shell.

We can imagine a system where the parser only recognizes the core language, and a set of exposed transformation rules would map the extended syntax which makes *es* feel like a shell, down to the core language. The *extend-syntax* [9] system for Scheme provides a good example of how to design such a mechanism, but it, like most other macro systems designed for Lisp-like languages, does not mesh well with the free-form syntax that has evolved for UNIX shells.

[3]This disabling depends on operating system support.

The current implementation of *es* has the undesirable property that all function calls cause the C stack to nest. In particular, tail calls consume stack space, something they could be optimized not to do. Therefore, properly tail recursive functions, such as `echo-nl` above, which a Scheme or ML programmer would expect to be equivalent to looping, have hidden costs. This is an implementation deficiency which we hope to remedy in the near future.

Es, in addition to being a good language for shell programming, is a good candidate for a use as an embeddable "scripting" language, along the lines of Tcl. *Es*, in fact, borrows much from Tcl – most notably the idea of passing around blocks of code as unparsed strings – and, since the requirements on the two languages are similar, it is not surprising that the syntaxes are so similar. *Es* has two advantages over most embedded languages: (1) the same code can be used by the shell or other programs, and many functions could be identical; and (2) it supports a wide variety of programming constructs, such as closures and exceptions. We are currently working on a "library" version of *es* which could be used stand-alone as a shell or linked in other programs, with or without shell features such as wildcard expansion or pipes.

Conclusions

There are two central ideas behind *es*. The first is that a system can be made more programmable by exposing its internals to manipulation by the user. By allowing spoofing of heretofore unmodifiable shell features, *es* gives its users great flexibility in tailoring their programming environment, in ways that earlier shells would have supported only with modification of shell source itself.

Second, *es* was designed to support a model of programming where code fragments could be treated as just one more form of data. This feature is often approximated in other shells by passing commands around as strings, but this approach requires resorting to baroque quoting rules, especially if the nesting of commands is several layers deep. In *es*, once a construct is surrounded by braces, it can be stored or passed to a program with no fear of mangling.

Es contains little that is completely new. It is a synthesis of the attributes we admire most from two shells – the venerable Bourne shell and Tom Duff's *rc* – and several programming languages, notably Scheme and Tcl. Where possible we tried to retain the simplicity of *es*'s predecessors, and in several cases, such as control flow constructs, we believe that we have simplified and generalized what was found in earlier shells.

We do not believe that *es* is the ultimate shell. It has a cumbersome and non-extensible syntax, the support for traditional shell notations forced some

unfortunate design decisions, and some of *es*'s features, such as exceptions and rich return values, do not interact as well with UNIX as we would like them to. Nonetheless, we think that *es* is successful as both a shell and a programming language, and would miss its features and extensibility if we were forced to revert to other shells.

Acknowledgements

We'd like to thank the many people who helped both with the development of *es* and the writing of this paper. Dave Hitz supplied essential advice on where to focus our efforts. Chris Siebenmann maintained the *es* mailing list and ftp distribution of the source. Donn Cave, Peter Ho, Noel Hunt, John Mackin, Bruce Perens, Steven Rezsutek, Rich Salz, Scott Schwartz, Alan Watson, and all other contributors to the list provided many suggestions, which along with a ferocious willingness to experiment with a not-ready-for-prime-time shell, were vital to *es*'s development. Finally, Susan Karp and Beth Mitcham read many drafts of this paper and put up with us while *es* was under development.

References

1. Brian W. Kernighan and Rob Pike, *The UNIX Programming Environment,* Prentice-Hall, 1984.
2. S. R. Bourne, "The UNIX Shell," *Bell Sys. Tech. J.*, vol. 57, no. 6, pp. 1971-1990, 1978.
3. Tom Duff, "Rc – A Shell for Plan 9 and Unix Systems," in *UKUUG Conference Proceedings*, pp. 21-33, Summer 1990.
4. William Clinger and Jonathan Rees (editors), *The Revised⁴ Report on the Algorithmic Language Scheme*, 1991.
5. Robin Milner, Mads Tofte, and Robert Harper, *The Definition of Standard ML,* MIT Press, 1990.
6. John Ousterhout, "Tcl: An Embeddable Command Language," in *Usenix Conference Proceedings*, pp. 133-146, Winter 1990.
7. Jon L. Bentley, *More Programming Pearls,* Addison-Welsey, 1988.
8. David R. Hanson, "Fast allocation and deallocation of memory based on object lifetimes," *Software—Practice and Experience*, vol. 20, no. 1, pp. 5-12, January, 1990.
9. R. Kent Dybvig, *The Scheme Programming Language,* Prentice-Hall, 1987.

Author Information

Paul Haahr is a computer scientist at Adobe Systems Incorporated where he works on font rendering technology. His interests include programming languages, window systems, and computer architecture. Paul received an A.B. in computer science from Princeton University in 1990. He can be reached by electronic mail at *haahr@adobe.com* or by surface mail at Adobe Systems Incorporated, 1585 Charleston Road, Mountain View, CA 94039.

Byron Rakitzis is a system programmer at Network Appliance Corporation, where he works on the design and implementation of their network file server. In his spare time he works on shells and window systems. His free-software contributions include a UNIX version of *rc*, the Plan 9 shell, and *pico*, a version of Gerard Holzmann's picture editor *popi* with code generators for SPARC and MIPS. He received an A.B. in Physics from Princeton University in 1990. He has two cats, Pooh-Bah and Goldilocks, who try to rule his home life. Byron can be reached at *byron@netapp.com* or at Network Appliance Corporation, 2901 Tasman Drive, Suite 208 Santa Clara, CA 95054.

Jgraph – A Filter for Plotting Graphs in PostScript

James S. Plank – Princeton University

ABSTRACT

Jgraph is a non-interactive filter for plotting two-dimensional scatter, line, and bar graphs in PostScript. It has also been used as a general-purpose drawing utility. Jgraph's strengths lie in its portability, flexibility, and integration into the UNIX environment. Jgraph is free software available on `netlib` or by anonymous `ftp`.

Introduction

Scientists in all disciplines frequently need to display information graphically on a variety of high-quality output devices. However, there is no standard tool on the UNIX platform that achieves this purpose. Although many software packages exist to facilitate plotting graphs, they all have limitations. Some are only available on certain machines; some can only be integrated into certain text processing systems; some require specific data formats; some are available only as part of colossal computing environments.

Jgraph attempts to provide a simple, yet flexible and powerful graph-plotting package. It is a filter that takes a description of a graph or graphs as input, and produces PostScript [1] as output. PostScript was chosen because it is a standard format for producing high-quality graphic output. PostScript can be viewed on the computer screen with a PostScript viewer like `gs`, printed directly on PostScript printers, or, when in *encapsulated PostScript (EPS)* format, embedded in a text or graphics processing system such as TeX, LaTeX, `troff`, Scribe, or Adobe Illustrator *88*. Moreover, since PostScript is in ASCII format, it can be stored on all hardware platforms and sent freely in all electronic mail systems. Jgraph has the option of producing either EPS or regular PostScript files.

Unlike almost all other graph-plotting packages, jgraph is *non-interactive*. In these days of "user-friendly" systems, this might be seen as a disadvantage, but the advantages of this decision are threefold. First, it allows jgraph to be used on all platforms, as it is not bound to specific terminal types, window systems or even operating systems. Second, it means that jgraph can solve one problem – graph plotting – and solve it well. This is in contrast to systems that provide their own editors, window systems, output viewers, etcetera, which are bound to conflict with the ones to which their users are accustomed. Finally, by being non-interactive, jgraph integrates well with the powerful utilities in UNIX (e.g., `sed`, `nawk`, `make`). Jgraph can be used in makefiles and as part of multistage UNIX pipes, and it can also execute shell commands from within its input. This gives the user a great deal of flexibility often absent from other graph-plotting packages.

Jgraph is free, portable, and well-documented. It is public-domain software that can be obtained over the internet either through `netlib`[1] or by anonymous `ftp`.[2] It is written in machine-independent C and comes with an 18-page manual and many example graphs, including those presented in this paper. It has been installed at over 60 locations under various operating systems, including all flavors of UNIX, as well as VMS and DOS. I am not aware of any environment containing a C compiler on which jgraph cannot be installed.

Jgraph Overview

Jgraph reads a description of graphs on the standard input and produces PostScript on the standard output. The input format is simple enough to let users create useful graphs as soon as they start learning the tool, yet flexible enough be general-purpose. Input consists of keywords followed by values, where a value is either a number, a string or another keyword. White space is ignored except within strings, so that input files may be indented for readability as in the figures below.

Appendix A gives a complete formal specification of the jgraph syntax. This section gives an overview of the salient features of jgraph, as well as a flavor for typical jgraph input and output files.

The major unit of jgraph's input is a *graph*: Users may specify any number of graphs for jgraph to plot on a page. Each graph consists of the following parts: *X* and *Y* axes, curves, strings, a title, a legend, and a position relative to other graphs.

The most important part of a graph are the *curves*. Users may specify any number of curves in a graph. Each curve consists of points, mark

[1] Send email with only the text: `send jgraph.shar from misc` to `netlib@ornl.gov`.

[2] `Ftp` to `princeton.edu`, and get the file `pub/jgraph.Z`.

attributes, line attributes, and a label. The points are (x, y) pairs that are plotted in the order given. Mark attributes define what gets plotted at each point (e.g., nothing, a circle, a box, text, or a bar-graph line to either axis). Line attributes define what kind of line gets plotted between points (e.g., none, solid, dotted). The label defines the legend entry for the curve.

Jgraph chooses defaults for all attributes, making simple graphs simple to create. The example in Figure 1 below shows the jgraph input and realized PostScript output of a simple graph with three curves. The topmost curve lets jgraph choose all the curve attributes–the only things specified are the points. The middle one plots triangles connected by a solid line, and the bottom one plots just a dashed line between the points. Jgraph sets up default values for all other parts of the graph.

```
newgraph

  newcurve
    pts 0 6  1 9  2 11  3 14  4 18 5 20

  newcurve
    marktype triangle
    linetype solid
    pts 0 3  1 4  2 7  3 9  4 10 5 13

  newcurve
    marktype none
    linetype dashed
    pts 0 0  1 2  2 3  3 5  4 6  5 9
```

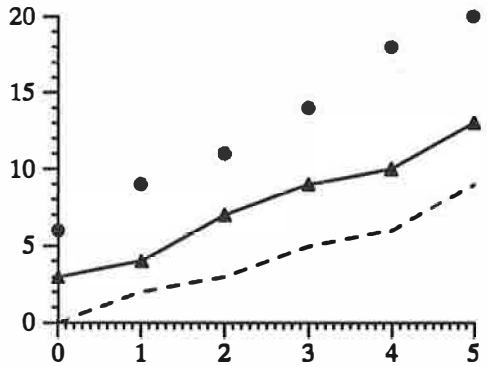


Figure 1: A simple jgraph input and output

Users may change the other graph attributes just as the curve attributes are changed in Figure 1. For example, for both axes, users may alter the axis size, maximum and minimum values, scaling (linear or logarithmic), location and spacing of hash marks, etcetera. Users also have control over the appearance and location of a graph's legends and titles, as well as the ability to plot arbitrary text strings anywhere on a graph.

Example 1: Figure 2 shows a more complex example graph in which many of the jgraph defaults are changed to get a desired effect. Here a label has been added to the x-axis, the y-axis is not drawn, and two strings are plotted with each bar: one to describe the bar, and one to state the bar's value. Note also the use of `copystring`, which copies the default values from the previous string. The tokens `copycurve` and `copygraph` are defined to do the same thing for curves and graphs.

```
newgraph
xaxis
  size 3 min 0 max 41
  mhash 1 (* Put 1 tick between hash marks *)
  hash_labels font Times-Italic
  label : Qualifications...

yaxis
  size 1.5 min 0
  nodraw (* Don't draw the y-axis *)

newcurve marktype ybar marksize 0 .6 fill .9
  pts 41 4   35 3   17 2   14 1

newstring
  hjl vjc  (* These define justification *)
  fontsize 9
  font Helvetica-Narrow
  x 1 y 4 : Led league in wins

(* Copystring copies the defaults *)
copystring y 3 : Played for first place team
copystring y 2 : Led league in ERA
copystring y 1 : Led league in K's

copystring x 42 y 4 : 41
copystring x 36 y 3 : 35
copystring x 18 y 2 : 17
copystring x 15 y 1 : 14

newstring
  hjr vjb
  fontsize 6
  x 41 y 0.1
  : Source: USA Today research
```

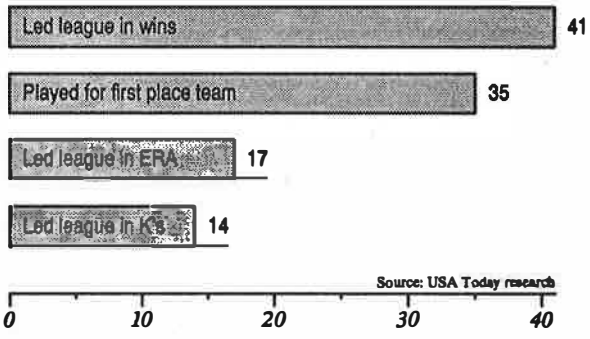


Figure 2: A more complex example

The treatment of strings is one of the elegant features of jgraph. All strings and string-like attributes are treated in the same manner. That is, strings, axis labels, the title, hash labels, legend entries and text marks are all manipulated by the same keywords. For example, there is a special string for each axis called `hash_labels`, which treats all hash labels on that axis as a unit. Thus, for example, the user can change the font on all the hash labels by changing the font of the string

`hash_labels`, as in Figure 2 above. Similarly, there is a special string for legends that controls all the legend entries as a unit.

Jgraph supports grayscales and color. Users can set the color or grayness of every part (strings, axes, lines, marks) of each graph. Figure 2 uses grayscale to shade each bar. Figure 4 shows a far more complex and effective use of grayscale in jgraph.

Accessing UNIX from jgraph

Jgraph's `include` and `shell` constructs allow users to include files and shell commands from within the jgraph input. This has two benefits. First, it enables the user to specify his or her own formats for data files and extract the data using UNIX utilities such as `sed`, `nawk`, or even C programs. This is in opposition to other programs which require data to be in a specific format.

Second, it frees jgraph from attempting to provide function plotting. Some graph-plotting packages include a facility to plot functions, usually something resembling a subset of a more general language (such as an expression evaluator in C with certain math libraries included, as in `gnuplot`). Jgraph omits any such facility, because users usually have their own resources for evaluating mathematical expressions which are more robust and powerful than those included in typical graph-plotting packages. The `shell` construct allows users to tap the powers of these resources in a simple and concise way.

Example 2: This example shows how to use the `shell` construct for both data extraction and function-plotting. In this example, the user has timed a program which sorts indexed records using a binary tree and would like to see how its running time compares with the theoretical running time of $O(n\log n)$, where n is the number of records. The program's output for varying values of n has been stored in the text file `data.txt`, which has the following format:

```
Number of records = 0       Time = 0
Number of records = 5000    Time = 2
Number of records = 10000   Time = 3
...
```

Thus the user can extract the data points for a graph of n versus time with a simple `nawk` script, which prints columns five and eight of `data.txt`. This is done in the first curve of the jgraph input in Figure 3. Next, the user wants to plot the function $n\log n/k$, where k is a constant that makes the data in `data.txt` fit the function. After determining a value of k=35000 the user can plot the function using the nawk script in the second curve of Figure 3. Thus, the `shell` construct of jgraph gives the user all the powers of the tools available under UNIX.

More complex graphs and drawings

Since jgraph allows users to control all parts of a graph and lets them arrange multiple graphs on a page, it can be used to plot arbitrarily complex graphs and even general purpose drawings. Since jgraph is non-interactive, it can be used as a back-end graphics language for making drawings that use graph constructs (such as axes and legends) or that have an iterative structure. Figure 4 is an example by Dave Wortman [11] which uses jgraph in such a way. The input file for this picture was created by a `nawk` script that processes data and emits jgraph output. "WYSIWYG" drawing tools like `xfig` or MacDraw are not suited for such tasks.

Figures 5 and 6 show further examples of complex, structured drawings that are straightforward to produce with jgraph but would be difficult to produce with a WYSIWYG tool. Figure 5, from [9] is a jgraph drawing which depicts processor communication over time. It makes use of jgraph's ability to plot axes and legends in a general-purpose drawing. Figure 6 is a jgraph drawing produced by a `nawk` script written by Adam Buchsbaum that takes a

```
xaxis size 2.5
  hash_labels font Helvetica
  label : Number of Indexed Records (N)
yaxis size 2.1
  label : Running time (seconds)
  hash_labels font Helvetica

newcurve
  marktype cross
  label : Data
  pts shell : nawk '{print $5, $8}' data.txt

newcurve
  marktype none linetype solid
  label : N log N / 35000
  pts shell : nawk \
     ' $5 != 0 { \
       print $5,  $5 * log($5) / 35000}' \
     data.txt
```

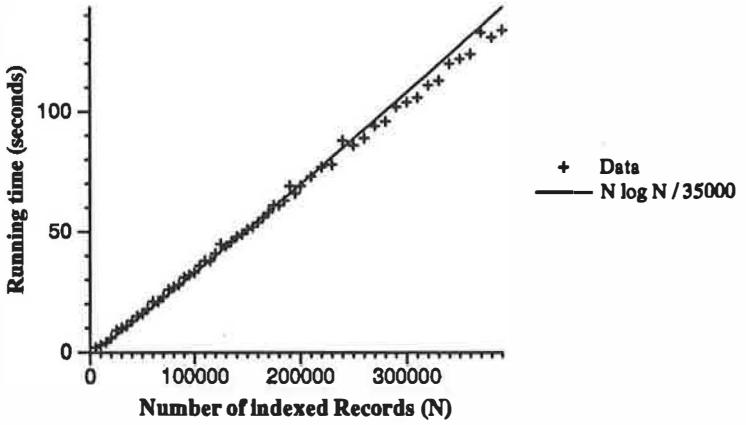


Figure 3: A more complex graph using the `shell` construct of jgraph

description of trees and produces jgraph output [4]. It treats jgraph as a convenient back-end graphics language.

Related Work

There are many programs that can be used for graph-plotting, ranging from simple filters like jgraph, to more complex software packages. The two standard UNIX programs for graph-plotting are `graph` [7] and `grap` [3]. Like jgraph, both are non-interactive filters, with `graph` producing output for the UNIX `plot` routine, and `grap` producing `pic` output for inclusion in `troff` documents.

`Graph` is a primitive program whose functionality comprises a restricted subset of jgraph's. `Grap` on the other hand, is a powerful tool with

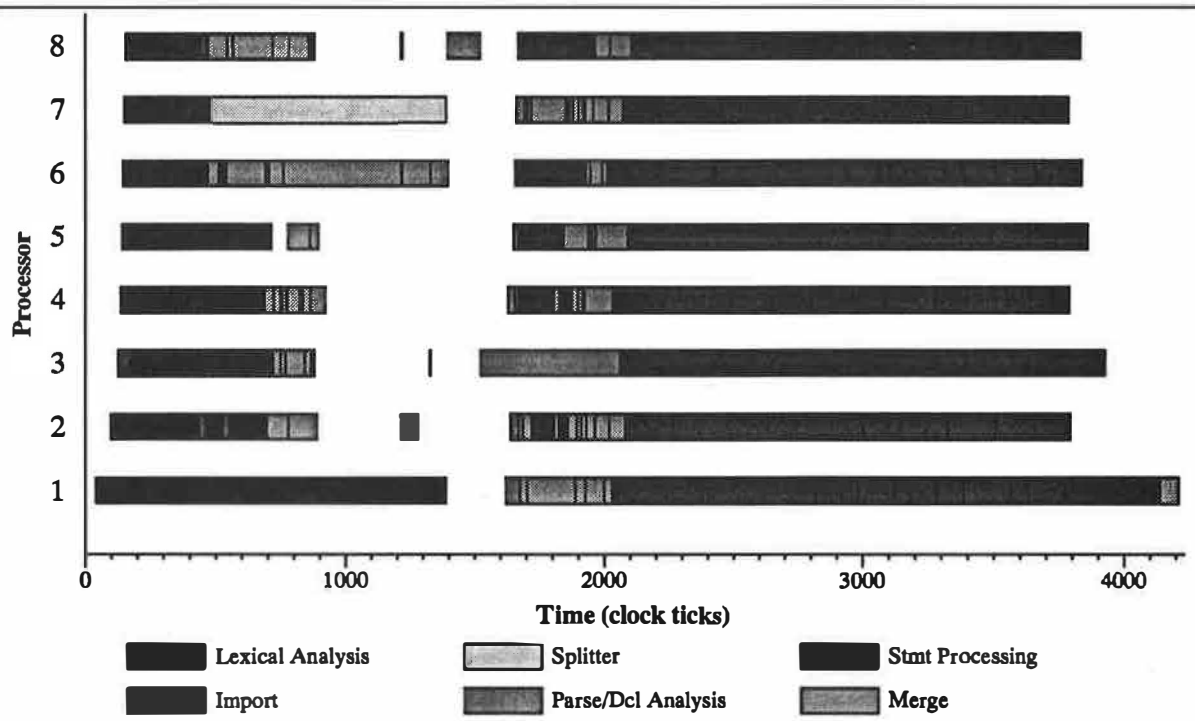


Figure 4: Results of a nawk-to-jgraph data processing program

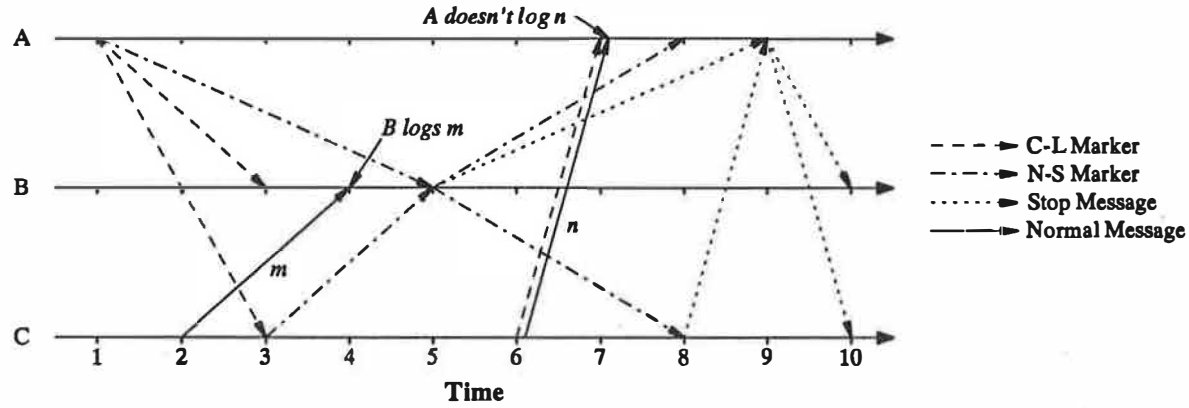


Figure 5: A jgraph drawing depicting processor communication

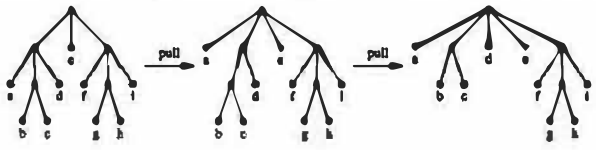


Figure 6: Results of a nawk-to-jgraph tree-drawing program

many of the same advantages as jgraph in terms of flexibility. However, `grap` was designed for use with `pic` and `troff` and therefore suffers from a few problems. First, `troff` and its family of programs were designed before the advent of today's high-quality PostScript printers. Therefore, the output of such programs, even when converted into PostScript, is often inferior to programs such as LaTeX, Scribe, or Adobe Illustrator *88*. Second, it is non-trivial to convert `grap` output into usable PostScript. For example, one can get TeX from `grap` by using the program `tpic`, and one can get printable PostScript from `grap` by using `psroff` or `psdit`. However, it is impossible to get encapsulated PostScript without hand-editing output files. Third, although `grap` is considered a standard part of UNIX, it is not available on all UNIX systems and is not easily ported to non-UNIX systems. Finally, most users (at least in the computer science community) use TeX and LaTeX instead of `troff` to process text, so they aren't prepared to take advantage of the flexibility offered by `grap`, as it relies on a thorough knowledge of `pic` and `troff` macros and constructs.

There are many interactive programs for drawing graphs: `Xgraph` [8], `Gnuplot` [10], and `Mathematica` [12] all run under UNIX. `Xgraph` is best described as `graph` with an Xwindows interface. Like `graph`, it suffers from a lack of flexibility. `Gnuplot` and `Mathematica` on the other hand are quite powerful, including facilities for function-plotting and 3D graph-plotting as well as for scatter, line, and bar graph plotting. Their interactiveness, however, makes them more cumbersome to use than jgraph for all but the simplest of plots, and in the tasks to which both they and jgraph are applicable, jgraph has the simpler interface.

There are other graph-plotting programs for non-UNIX systems, such as CricketGraph [6] and Excel [5] for the Macintosh and other personal computers, and RS/1 [2], a massive data processing package available on VMS. None of these are portable to Unix systems, nor are any of them free software.

Acknowledgements

The author would like to thank Matt Blaze, Heather Booth, Adam Buchsbaum, and Norman Ramsey for their comments concerning this paper, Kent Landfield and Reed Wade for helping to distribute the software, and Dave Wortman for creating beautiful jgraph drawings. The author has been funded in part by an AT&T fellowship.

References

[1] Adobe Systems Incorporated. *PostScript Language Reference Manual.* Addison-Wesley, Reading, Massachusetts, 1985.

[2] Bolt, Beranek, and Newman. *RS/1 Users Guide.* BBN Software Product Corporation, 1987.

[3] Jon L. Bentley and Brian W. Kernighan. GRAP – A language for typesetting graphs tutorial and user manual. Technical Report #114, AT&T Bell Laboratories, December 1984.

[4] Adam L. Buchsbaum and Robert E. Tarjan. Confluently persistent deques via data structural bootstrapping. In *Proceedings of the 4th ACM-SIAM Symposium on Discrete Algorithms*, January 1993.

[5] Douglas F. Cobb, Allan McGuffy, and Mark Dodge. *Microsoft Excel 3 companion.* Microsoft Press, Redmond, Washingon, 1991.

[6] *Desktop Software Guide.* Computer Associates International Inc., Islandia, NY, 1992.

[7] graph – draw a graph. Unix man page, 1983.

[8] David Harrison. xgraph – draw a graph on an x11 display. Unix man page, 1989.

[9] Kai Li, Jeffrey F. Naughton, and James S. Plank. An efficient checkpointing method for multicomputers with wormhole routing. *International Journal of Parallel Processing*, 20(3), June 1992.

[10] Thomas Williams. gnuplot – an interactive plotting program. Unix man page, 1990.

[11] David B. Wortman and Michael D. Junkin. A concurrent compiler for Modula-2+. *ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, in SIGPLAN Notices*, 27(7):68-81, July 1992.

[12] Stephen Wolfram. *Mathematica, A System for Doing Mathematics by Computer.* Addison-Wesley, Redwood City, California, 1988.

Author Information

Jim Plank is an assistant professor at the University of Tennessee in Knoxville. He received his PhD from Princeton University in December, 1992. His research area is general fault-tolerance in parallel and distributed computing. Jgraph is a hobby. He can be reached at: Department of Computer Science; University of Tennessee; Knoxville, TN 37966 or by Email at plank@cs.utk.edu.

APPENDIX A: Formal Syntax of Jgraph

```
<top-level> :=   % Top Level commands
   <nil> |
   <top-level>* |

   newgraph <graph> |  % Choose/edit graphs
   graph <int> <graph> |
   copygraph [<int>] <graph> |

   newpage |           % General layout commands
   bbox <int> <int> <int> <int> |
   X [<float>] | Y [<float>] |
   preamble <file> | epilogue <file>

<graph> :=
   <nil> |
   <graph>* |

   newcurve <curve> |       % Edit curves
   curve <int> <curve> |
   copycurve [<int>] <curve> |
   newline <curve> |

   xaxis <axis> |           % Edit the attributes
   yaxis <axis> |           % of each axis

   newstring <string> |     % Edit and plot
   string <int> <string> |  % arbitrary strings
   copystring [<int>) <string> |

   title <string> |      % Edit the graph's title
   legend <legend> |     % Edit the legend

   border | noborder |   % Draw a border around the graph
   clip | noclip |       % Clip inside this border

   x_translate [<float>] |  % The graph's position
   y_translate [<float>]    % relative to other graphs

<curve> :=        % These commands allow the user to
   <nil> |        % enter curve points and attributes
   <curve>* |   % enter curve points and attributes

   pts [<float> <float>]* |  %Point definitions
   x_epts [<float> <float> <float> <float>]* |
   y_epts [<float> <float> <float> <float>]* |

   marktype <marktype> |     % Mark definitions
   marksize [<float>] [<float>] |
   mrotate [<float>] |
   gmarks [<float> <float>]* |
   postscript <file> |
   fill [<float>] | cfill [<float> <float> <float>] |

   linetype <linetype> |      % Line definitions
   linethickness [<float>] |
   glines [<float>]* |
   gray [<float>] | color [<float> <float> <float>] |
   pfill [<float>] | pcfill [<float> <float> <float>] |
   bezier | nobezier |

                              % Arrowheads on lines
   larrow | rarrow | nolarrow | norarrow |
   larrows | rarrows | nolarrows | norarrows |
   asize [<float>] [<float>] |
   afill [<float>] | acfill [<float> <float> <float>] |

   label <string>   % The legend entry
   clip | noclip |  % Whether to show points outside the
                     % max and min axis values

<marktype> :=              % Different types of marks
   none | general |
   circle | box | diamond | triangle | x |
   cross | ellipse | xbar | ybar | text | postscript

<linetype> :=              % Different types of lines
   none | general |
   solid | dotted | dashed | longdash |
   dotdash | dotdotdash | dotdotdashdash

<string> :=     % These commands let the user change the
   <nil> |      % appearance and location of any string
   <string>* |

   : <chars> |
   x [<float>] | y [<float>] |
   rotate [<float>] |
   hjl | hjr | hjc | vjt | vjb | vjc |  % Justification
   font <fontname> | fontsize [<float>] |
   linesep [<float>] |
   lgray [<float>] | lcolor [<float> <float> <float>] |

<axis> :=       % These commands let the user edit the
   <nil> |      % attributes of an axis
   <axis>* |

   linear | log | log_base [<float>] |
   min [<float>] | max [<float>] | size [<float>] |

   label <string> |

   draw | nodraw |
   gray [<float>] | color [<float> <float> <float>] |

   draw_axis | nodraw_axis | draw_at [<float>] |
   draw_axis_label | nodraw_axis_label |
   grid_lines | no_grid_lines |
   mgrid_lines | no_mgrid_lines |

   hash [<float>] |    % These commands let the user change
   shash [<float>] |   % the appearance of the hash marks
   mhash [<int>] |     % and labels
   precision [<int>] |
   hash_at [<float>] | mhash_at [<float>] |
   hash_label <hash_label> |
   hash_labels <string> |
   hash_scale [<float>] |
   draw_hash_marks | nodraw_hash_marks |
   draw_hash_labels | nodraw_hash_labels |
   draw_hash_marks_at [<float>] |
   draw_hash_labels_at [<float>] |
   auto_hash_marks | no_auto_hash_marks |
   auto_hash_labels | no_auto_hash_labels

<hash_label> :=    % These commands let the user create
   <nil> |         % his or her own hash labels
   <hash_label>* |
   at [<float>] |
   : <chars>


<legend> :=        % These commands govern the legend
   <nil> |
   <legend>* |

   on | off | left | right |        % Location
   top | bottom | custom |
   x [<float>] | y [<float>] |

   linelength [<float>] |
   linebreak [<float>] |
   midspace [<float>] |

   defaults <string>


% Other tokens are obvious -- e.g. <int> and <float>.
% At any point in the input, you may have:

include <file>    % Include the contents of the <file>.
shell : <chars>   % Execute the <chars> as a shell command
                  % and include the contents of stdout.

(* <chars> *)     % Comments, which are ignored
```

Faster AFS

Michael T. Stolarchuk – University of Michigan

ABSTRACT

The AFS Cache Manager fetches files from the AFS file server, and caches them into a local file system. Given this model, users expect reads of locally cached files to perform at local file system rates. However, read performance of the AFS cached files is half the read performance of the local file system. This paper discusses the reasons for the large performance difference, and the modifications made to AFS so that reads of locally cached files perform within 10% of the performance of the local file system.

Introduction

At the Center for Information Technology Integration (CITI), we developed a UNIX-based, AppleShare server [1] to support the file system needs of the University of Michigan's large Macintosh community. The AppleShare server will integrate the University's Macintosh users into the planned campus-wide, AFS-based Institutional File System [2]. Therefore, performance is critical.

During the development of the AppleShare server, we studied its performance on three different file systems: UFS (Berkeley UNIX File System) [6], JFS (IBM's AIX Journaling File System), and AFS (Transarc's AFS File System) [3, 7]. We were surprised to find that read operations in UFS and JFS ran twice as fast as those in AFS.

Although we initially thought the performance difference was due to factors such as network performance and file server platforms, measurements pointed to the slow speed of the AFS reads. We were forced to investigate the difference.

The AppleShare server is an ordinary application, built on top of a socket interface to the AppleTalk address family running native in a Berkeley UNIX kernel. The server uses UNIX file system calls (open, close, read, write, etc.) to process client requests. When a Macintosh wants to read, the AppleShare server opens a UNIX file, issues a read request, and returns the data.

AFS clients cache files. When an application references a file for the first time, the file is fetched from the server into a cache on the client's underlying native file system, e.g., UFS. This bounds the performance of AFS to that of the local file system. Comparing read times from AFS to read times from the local file system measures the AFS overhead. We have reviewed the implementation of the AFS read operation, reorganized the code, and added appropriate hooks solely for performance. Our new implementation runs much faster – the AFS overhead has been reduced from 100% to about 10%.

The next section sets the context for the remainder of the paper by describing the methodology we used to minimize the AFS read overhead. The remainder of the paper gives the details of that process and describes the performance of the results – Faster AFS.

Methodology

This paper describes finding and fixing performance problems in AFS. Unlike traditional techniques of rewriting the code based on the results of measurement, we used a skeletal read as the basis for our performance improvements.

We assumed that once a file was cached on the local file system, reading the file through AFS would be as fast as reading the file from the local file system. However, we measured the AFS read as being much slower than the local file system read. As we couldn't explain this difference, our first goal was to understand why the code behaved differently from our expectations.

Profiling the AFS Read

Our short term goal was to improve the AFS read performance using profiling. Therefore, we built a simple benchmark to measure the file system's read performance and used profiling to count the instructions. We found several inefficiencies in the implementation, and made minor code changes to remove them. The read performance was then measured again. After several of these profiling sessions, we converged on an AFS read that performed faster than the original version. Benchmarks and profiling showed the overhead to be some 50% for 4K reads on an IBM RT, when compared to the unmodified version of the AFS client.

Because most code segments now executed with equal frequency, profiling no longer provided clues for performance improvements. However, the cost of performing all the AFS read requirements (such as cache consistency checking) was higher then we originally thought and the cost of performing the actual read to the local file system was lower than expected.

Even with contrary evidence from the profiling, we believed reading a locally cached file through AFS should be as fast as reading the local file.

However we were willing to amend our performance goals, lowering our expectations to an arbitrary 10% overhead (from no overhead). We believed users would be willing to tolerate 10% overhead.

Incorporating the Short Path Test Case

Since we already amended our goal once, and since the list of AFS read responsibilities was large, we questioned whether the 10% goal was possible. To test this we started by determining the smallest overhead. We made the shortest path between the benchmark and the local file system. We then modified the AFS client read by making a short cut to the local file system, bypassing all the read requirements. The short path, for example, breaks cache consistency. The test read delivered real file data, and we measured the time to reach the underlying file system.

We expected low overhead, around one to two percent. We discovered instead an overhead of almost 10%. As only a few AFS-related instructions were in the test read, the local file system operated much more quickly than we thought. This left little additional room to add back the requirements of the original AFS read procedure and still meet our goal of 10% overhead.

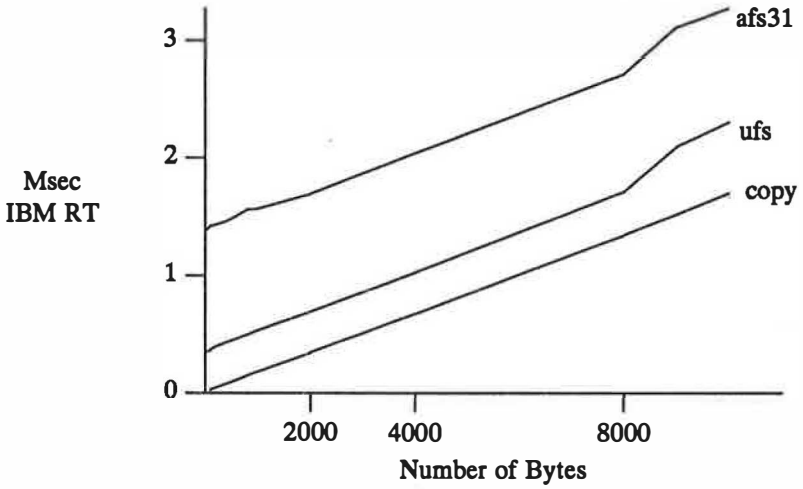
It was clear that we could not completely bypass the read requirements. However, we questioned whether it was necessary for the read to satisfy each requirement on every execution. That would then be our approach – discover a method of alternative paths through the read procedure that would result in overall performance gains.

Performance Measurements

The performance problem is illustrated by comparing the read times of AFS to the local file system. The local file system measured is the same one used by AFS to cache files fetched from the server. These measurements show that applications reading from locally cached AFS files perform poorly compared to applications reading data from the local file system.

Benchmark

The benchmark opens a file once, then reads from the beginning of a file either in AFS or the local file system. The amount of data read is varied. To help measure the executed code path, the benchmark reads the same seek location in each iteration, without performing any physical disk I/O.



	IBM RT		
Bytes	copy	ufs	afs31
100	0.028	0.336	1.056
1000	0.182	0.354	1.031
2000	0.347	0.349	0.997
4000	0.675	0.350	1.010
8000	1.342	0.365	1.001

Figure 1: The graph shows measurements of the AFS 3.1 (afs31) read, the Berkeley Fast File System (ufs) read, and memory to memory copies (copy) on the IBM RT. The table lists the time spent in milliseconds to copy the data from the read operation and describes the overhead for the other components by listing the additional time spent above the copy. The 'ufs' column describes the time spent performing local file system operations, and the 'afs31' column describes the time spent within the AFS 3.1 Cache Manager.

We only measured short data read requests, up to 8K. Most processes read using the standard I/O library, which issues reads at the basic block size of the file system, usually 8K or smaller. Applications not using standard I/O rarely issue read requests for more than 8K.

The benchmark measurements include copy times. A large component of the read time of local file systems is used to copy the data from the kernel address space to the user process. This also allows us to separate the static service time of the local file system, from the time used to move data.

Measurements

Figure 1 shows the data generated from an IBM RT, about 2 MIPS. Figure 2 show the data generated from an IBM RS/6000 520, about 20 MHz (since the RS/6000 is a superscalar machine, it's difficult to characterize MIPS performance).

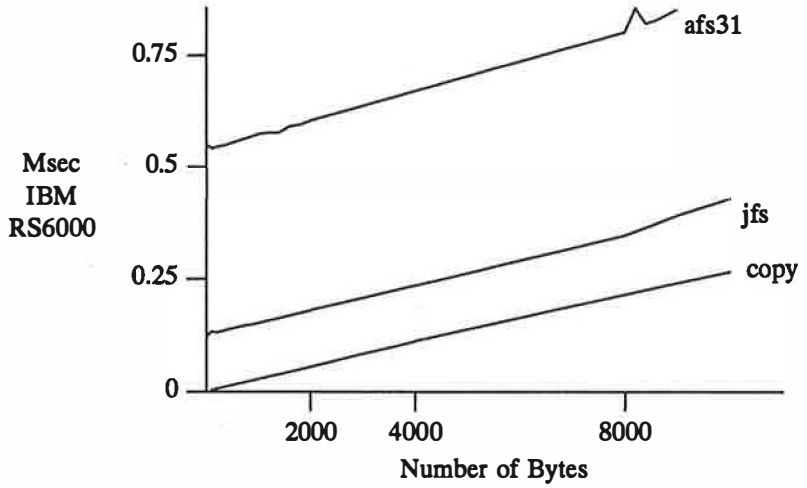
Discussion

We were surprised to find the AFS overhead of 100% for 4K reads on the IBM RT. We thought the overhead would be near zero since the file's data was on the local file system. After all, AFS only has to issue the read to the local file system.

The measured AFS overhead is high partially due to the benchmark, which doesn't cause any I/O. Although this may not seem representative of general use, some of our AFS clients have significant (128 Megabyte) memory caches. If the benchmarks had caused actual disk I/O, the service time of the reads would be much higher.

The tables from figures 1 and 2 show that the AFS overhead stays relatively static, as do the read times of the local file system. On the RT, for example, the static overhead of UFS to read one block from the local file system is about the time to move 2K of data. The static overhead of the AFS Cache Manager read is three times larger.

If we could characterize the size of the read requests issued by applications, we could infer the typical AFS overhead. According to Zhou [8], 70% of applications perform read requests for 4K or less. (Apparently the block size of the file system measured by Zhou is 4K). Determining the size of the typical read requests is difficult, since most reads come from from the standard I/O library, and the



IBM RS/6000 520			
Bytes	copy	jfs	afs31
100	0.004	0.128	0.409
1000	0.028	0.124	0.421
2000	0.055	0.124	0.424
4000	0.112	0.122	0.435
8000	0.215	0.131	0.456

Figure 2: The graph shows measurements of the AFS 3.1 (afs31) read, the AIX 3.1 Journaling File System (jfs) read, and memory to memory copies (copy) on the IBM RS/6000 520. The table lists the time spent in milliseconds to copy the data from the read operation and describes the overhead for the other components by listing the additional time spent above the copy. The 'jfs' column describes the time spent performing local file system operations, and the 'afs31' column describes the time spent within the AFS 3.1 Cache Manager.

standard I/O library determines its block size from the file system. We can tell the block size by looking at specific file systems. On the IBM RT's local file system (UFS) the block size is 8K, for a typical AFS overhead of 60%. The IBM RS/6000's local file system (JFS) has a block size of 4K, for a typical AFS overhead of about 180%.

The AFS 3.1 Cache Manager Read Requirements

To explore the cause of high overhead, we profiled the kernel to determine what AFS is doing when it controls the processor. The result of the profiles is used to cost the read requirements. The AFS read requirements are described here by using the source code, and an understanding of the goals of the AFS read. The following sections describe the major requirements, in decreasing cost order.

Cache Consistency

The data in the cached file must represent up-to-date information. The AFS Cache Manager uses a lazy policy to determine if the AFS file is out of date. Before any data of an AFS file is referenced, the file is checked for cache consistency. Early in each and every read operation, the AFS Cache Manager tests to determine whether the cached data is up-to-date. The test is straightforward but does involve several different comparisons. For example, if the file is from a read-only volume, it is assumed to be up-to-date.

If the file is within a read-write volume, then it is consistent if a "callback promise" exists. A callback is a promise made by the file server to inform the client if a file's status changes. Callbacks in AFS 3.1 have limited duration, depending on the number of concurrent users of the AFS file. The duration is currently quantized, with a maximum duration of 4 hours, for 0 to 7 users, and a minimum of 7 minutes, for over 64 users.

Chunk Location

Every AFS file is managed by the AFS Cache Manager as chunks in the local cache. Files that do not fit into a chunk are broken into multiple chunks. Chunks are fixed size and implemented as a file in the local file system. Only the chunks currently referenced by the application need to be in the cache. This means there are many chunks for one large AFS file, implying a mapping from an AFS file *and offset* into a chunk.

In the AFS Cache Manager, that mapping is performed through a hashed list of file identifiers. The AFS file is identified by a set of numbers, the File Identifier (FID), which consists of cell number, volume number, vnode number, and a "uniquifier."

Chunk Isolation

Although an AFS file is managed in chunks, the application is isolated from the implementation of chunks. If the user process requests data from an AFS file, and the request spans several different chunks, the read code must break up the original request into several smaller requests, each completely satisfied from one chunk.

The vnode interface [4] of the local file system reads and writes chunks, allowing the AFS Cache Manager to be relatively portable. This implementation strategy allows us to determine the overhead of the AFS Cache Manager reads, by comparing the performance of the local file system and the performance of AFS reads.

Early Return

If a chunk is not within the local cache, the read procedure must request its contents from the AFS file server. If the application requests a small number of bytes at the front of a file, read returns to the application when part of the chunk is filled.

The AFS Cache Manager keeps track of the highest byte retrieved from a file server for a chunk. A flag in the chunk indicates when the chunk is actively being fetched. After the read locates the related chunk, it checks to see if the data is currently being fetched. If it is, then the read waits until the desired data is received.

This implementation is straightforward, except that the user process waits for the chunk to fill. Therefore, other process must be filling the chunk. AFS typically configures two background processes during early system initialization to perform such activities. If a chunk needs to be fetched, the AFS Cache Manager has code to attempt to perform the fetch through the background processes, with the hope that the actual reading process can return early.

Prefetching

The AFS Cache Manager tries to hide some network and server latency by queuing fetch requests for the next chunks of a file. When the read is nearly completed, the background daemon receives a request to fill the next chunk.

A Faster Implementation

After spending time profiling the AFS code, and removing obvious bottlenecks, we were still paying 50% overhead for 4K read requests on the IBM RT. We needed another approach to meet our goal of 10%.

We considered completely rewriting the AFS read operation, but decided this approach would risk injecting bugs, and wouldn't necessarily improve performance. Portability would also be an issue. The current read routine has been ported to a significant number of different platforms, and has the #ifdefs to prove it. We didn't have all the different platforms, and didn't want to guess at the #ifdefs needed. We decided not to rewrite the read code, and began to question whether we could reduce to the 10% overhead using any evolutionary strategy.

To gain some insight into a new approach, we started by determining whether the 10% goal was possible. We constructed an AFS read with a short cut to the local file system, that bypassed all of the read requirements described in the previous section. We placed a call to the local file system at the first executable statement within the AFS read procedure. We then measured the overhead using just the short cut. We included a toggle to turn the short cut on and off. A significant amount of AFS code still exists even in the short cut path. (The entry into the AFS code for a read request is through the rdwr vnodeop, while the call to the local file system is made in an AFS read routine for non-directory files). Using the benchmark measurement, 200 byte reads already performed at 12% overhead.

To meet our goal of 10% overhead, we couldn't selectively add back requirements. Each requirement described would add some 5-10% overhead. Instead, we reorganized the AFS code to allow the new short cut to meet the performance objective.

We decided to use the short cut only some of the time, creating two primary paths through the read procedure – the short path and the worst case path. We wanted the short path to become the commonly executed path through the new read procedure. This strategy allows us to reuse existing read code. If we can't execute the short cut to the local file system, we can execute the old read procedure. This improves performance without making a significant investment in new code.

Meeting the Requirements

To meet our performance objective, the short path needs to become the commonly executed path through the read procedure. In the test read, we used a toggle to decide whether to use the short path. In Faster AFS, we use test conditions to determine whether we can execute the short path. When the test conditions fail, we execute the long path to the read procedure. Since the long path is still available, these test conditions only need to direct the most common read requests through the short path.

The short path needs the parameters to pass to the read of the local file system. We call those parameters, along with other data, the "hint" [5]. The hint is populated while executing the long path through the read procedure. On the next read call, the short cut tests fields within the hint to see if it can use the short path. The contents of the hint also depend on the requirements of the read procedure.

The hint prejudices the performance of the read code. Some applications will perform well, others will see only minor improvement. Our hint is prejudiced towards applications that continue reading within the same chunk. Since the chunk size can be varied on a client-to-client basis, the size of the chunk could be tuned for specific workstations.

We now review each of the requirements described previously, to determine what data needs to be included in the hint, and also to determine what tests need to be made to perform the short cut.

Cache Consistency

In AFS 3.1, the code must constantly check to ensure the cached file is up-to-date. The callback includes a timeout value that is compared with the current time. When the time out is passed, the callback expires. Since a callback is only tested when it is necessary to check the validity of a file, this is a lazy policy. There is no central management of all the callbacks of the entire pool of cached files.

In some early performance analysis of AFS 3.1, we determined the ratio of AFS system calls to callback validity tests – seven callback validity tests for each AFS system call. To determine how often these validity tests were performing valuable work, we needed additional insight into the distribution of expiration times. We extracted callback timeout values from the AFS Cache Manager. Most values expired far in the future. Entries with the smallest timeouts were usually a few minutes from expiring. We performed a very limited study of the callback expirations, only workstations used for development were studied. Due to the bursty activity of the machines studied, callback timeouts were clustered around many different times. However, most of the callbacks did not expire for at least several minutes.

Since most timeouts expired far into the future, very few of the validity tests made repeatedly by the AFS client were performing valuable work. This situation suggested that the expiration test should be performed using some other policy. Therefore, we reimplemented cache consistency to manage the callbacks actively.

We use a doubly linked list sorted by timeout to collect the callback promises. Once a second, we test the top element to determine whether its callback should expire. If so, we modify the associated file to reflect the expiration. If the server delivers its callback promise to the client, and the callback of the file expires, it is removed from the list.

For any AFS file protocol request that returns a callback, the timeout is computed, and the vcache entry is sorted on the callback-expiration list. We currently search for the correct insertion point by starting at the end of the linked list (furthest into the future) and then move towards the beginning of the list (towards current time), on the assumption that returned callback timeouts tend to be distant events rather than immediate events.

Chunk Location

The AFS Cache Manager searches for the chunk associated with the file request at each read. The file and offset request are mapped to a chunk reference. Even though the chunk entries are hashed, the search is expensive. To keep from

searching the list at each read, the long path saves the last chunk referenced as part of the hint. That chunk is typically 64K. A relatively large number of sequential reads can be satisfied by that one chunk. The chunk size is configured at AFS Cache Manager initialization; if there isn't enough locality of reference in 64K chunks, the chunk size can be increased.

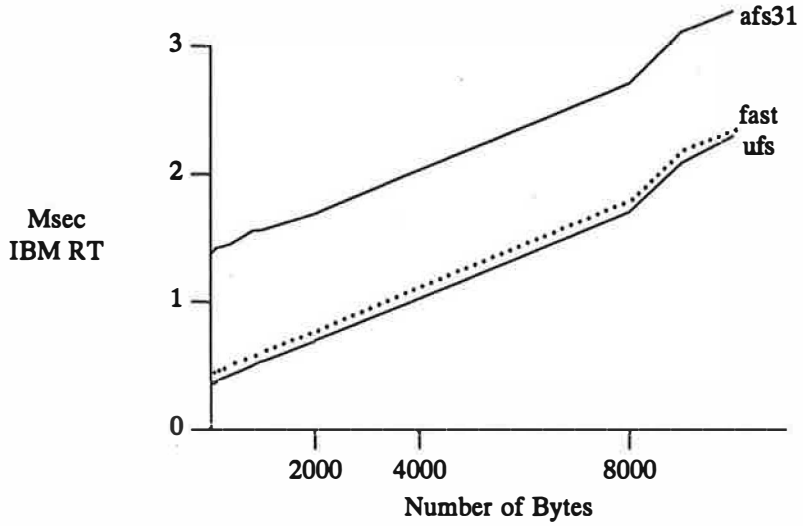
The short path needs to check if the hint is describing the file currently being read. We can compare the file identifier of the file being read and the chunk. If they match, the hint is describing the correct file. We can, however, construct a much simpler test. The structure describing the AFS file and the chunk can be stamped with a 32-bit value. If the two values match, the file ids are considered equal. The 32-bit value is a monotonically increasing number, incremented once for every tuple we want to relate. The stamp is computed once for the file/chunk pair, and the file and chunk structures are stamped with the same value. The stamp can then test for the file match in one comparison.

The short path also needs to ensure that the read request is requesting this particular chunk. Chunks are commonly described by chunk numbers, while read operations request offsets. To make the test in the short path simple, when we save the chunk reference we will also compute the offset of the chunk in the file and make it part of the hint.

Chunk Isolation

The short path is used only when the request is totally contained within one chunk. By testing to see if the user's request can be satisfied within one chunk, we don't have to be concerned about providing support for isolation directly. We use the hint when the user's request is within one chunk and ignore it otherwise. As mentioned earlier, if we find too many requests processed by the long path, we simply increase the chunk size. The chunk size can be modified only at system startup during AFS Cache Manager initialization.

No additional code is included in the short path to process reads that cross chunk boundaries. Instead, the short code depends on the existing long code path to process long read requests. This



	IBM RT				
Bytes	ufs	afs31	overhead	fast	overhead
100	0.36	1.05	290%	0.088	22.8%
1000	0.53	1.03	190%	0.069	12.8%
2000	0.69	0.99	140%	0.068	9.7%
4000	1.02	1.01	98%	0.089	8.6%
8000	1.70	1.00	58%	0.083	4.8%

Figure 3: The graphs shows performance measurements of the IBM RT for the Berkeley Fast File System (ufs), the AFS 3.1 Cache Manager (afs31), and the short path through the AFS 3.1 Cache Manager (fast). The table compares the read performance, measured in milliseconds, of these three implementations. The table also displays overhead for 'afs31' and 'fast.' Overhead is computed by dividing the observed AFS Cache Manager performance by the local file system performance.

method allows the short code path to focus on providing performance for typical applications, while still correctly processing large read requests. Additionally, the larger read requests can tolerate longer overhead, due to the time spent processing the request in the local file system. We wouldn't realize the same significant performance gains by decreasing overhead on the large read requests.

To perform the test quickly in the short path, we need to test the bounds of the chunk against the user request. When the request is completely contained within the chunk, the local file system can service the request directly. When we save the chunk reference in the long path, we also compute the bounds of the chunk as offsets from zero. The read request uses the same units, making the comparison to use the short path simple.

Early Return

The hint can't be populated until the chunk is completely filled. If the AFS Cache Manager sends a read request contained within one chunk to the local file system layer while the chunk is still being filled, the read could return the data from the partially filled chunk. However, the read would return without satisfying the entire request. The user application is unlikely to have the additional code to retry for additional data intended to be within that chunk. To preclude this event, we can't use the short path for chunks that are currently requested from the file server. We implement this by not populating the hint until the chunk is completely filled.

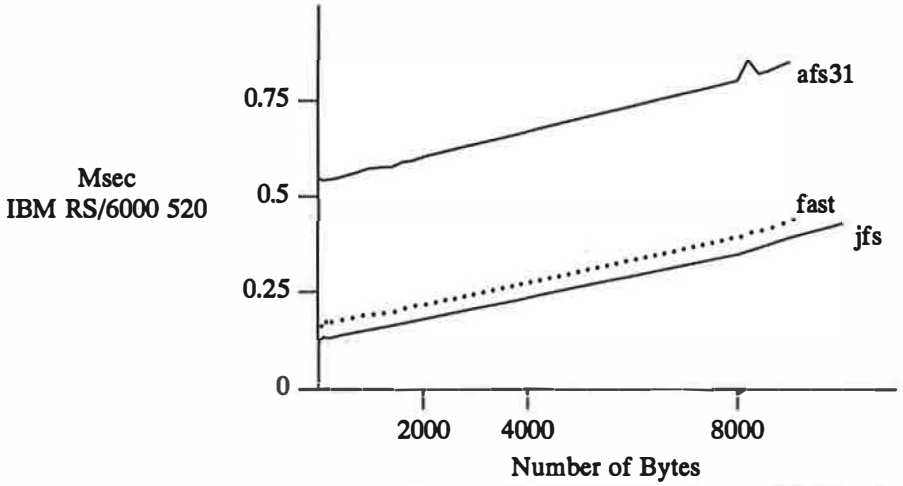
Prefetching

We currently have no additional code to support prefetch. We depend on the existing code in AFS 3.1 to perform some prefetch of chunks.

Measurements

The performance of the Faster AFS modifications appears in figures 3 and 4. These figures represent approximately 9% overhead for the IBM RT, and 15% overhead for the IBM RS/6000 both for 4K reads.

The reason the IBM RS/6000 incurs a larger overhead for the fast reads than does the IBM RT is unclear. It may be due to the longer time to perform indirect subroutine calls. The IBM RS/6000 has housekeeping to perform, which requires about 10 instructions for indirect function calls. This housekeeping may also help explain some of the larger overhead values for AFS 3.1 Cache Manager.



IBM RS/6000 520					
Bytes	jfs	afs31	overhead	fast	overhead
100	0.13	0.40	309%	0.035	26.5%
1000	0.15	0.42	276%	0.034	22.3%
2000	0.17	0.42	236%	0.034	18.9%
4000	0.23	0.43	185%	0.035	14.9%
8000	0.34	0.45	131%	0.042	12.1%

Figure 4: The graphs show performance measurements of the IBM RS/6000 520 for the AIX 3.1 Journaling File System (jfs), the AFS 3.1 Cache Manager (afs31) and the short path through the AFS 3.1 Cache Manager (fast). The table compares the read performance, measured in milliseconds, of these three implementations. The table also displays overhead for 'afs31' and 'fast.' Overhead is computed by dividing the observed AFS Cache Manager performance by the local file system performance.

Additional Concerns

Since the hints leave vnodes open, Faster AFS can act as a resource hog. Because the number of AFS stat structure entries is limited, and because each AFS stat structure can potentially have one open vnode (as a hint), large numbers of vnodes could be left open. AFS bounds the number of vcache entries, however, and this simple mechanism keeps the number of open vnodes low. As the AFS stat structures in the AFS stat pool are reused, open vnodes (hints) are freed.

A large number of in-use vnodes can be a concern in systems with very limited, statically allocated vnodes. In these situations, a pool allocator for vnode references can be used. The hint could save the pool reference, along with an ownership stamp. The short path would then also need to test ownership of the vnode reference in the pool by testing the stamp.

Currently the code surrounding the hint promotion and clearing does not lock the contents of the hint structure. We avoided locks not due to performance issues, but rather due to possible deadlock conditions resulting from the server delivering callback promises. We consider this issue open, and need to spend more time to determine a good solution.

Additional Work

Additional work could improve local caching, both by using faster caches, and better cache replacement policies. We considered making changes to the underlying file system, or making a specialized caching file system. We decided a memory cache would work well, and Transarc had already supplied a simple and effective memory cache. We plan to study one client with 128 megabytes of real storage, using a memory cache, and chunk sizes of 64K to see the upper boundaries of performance. We have also considered using cost-based cache replacement policies, to discard data easily recreated.

Because reads are the most heavily used operation, they were our our immediate concern. The same modifications are equally suited for write operations. More needs to be done to the short path to better adapt to the needs of program loading. For load-on-demand paged style text, hinting provides some benefit; for text-shared executables, it is likely the load request will span chunks. The hint only helps for requests within one chunk, text-shared program loading doesn't benefit from this hinting mechanism. This may not be an issue since the kernel read requests for text-shared program loads are often done for the complete contents of text (and data), and therefore the large static overhead of AFS is only paid once.

There may be some opportunity to use the same mechanism in other AFS operations. The AFS path lookup would seem to be a likely candidate, but AFS already incorporates an additional caching mechanism for directories.

Additional work could be done to determine the impact of multiple hints. Background daemons can populate a second hint while performing prefetch operations on behalf of an application. When the application performs a read on a prefetched chunk, the second hint can be used to select the short path. Multiple hints may not improve performance, however, since additional tests are added to the longer path through the AFS read procedure.

The AFS 3.1 Cache Manager is similar to the Open Software Foundation's (OSF) Distributed File System (DFS) Cache Manager. The same changes made to improve the performance of the AFS Cache Manager will also improve the performance of the DFS Cache Manager. Cache consistency is managed differently in DFS, using a token manager to coordinate read and write access. The token manager side steps the callback issues central to this paper, using an even more active policy than this paper does.

Although this work was initiated by the AppleShare server, we haven't yet measured the impact of Faster AFS on the performance of Macintoshes.

Conclusions

Function and interface aren't sufficient to characterize a service, costs are also important. When costs are left out, users of a service create a cost based on their intuition (from previous uses of the service). This intuitive view will likely be inaccurate. The performance of AFS suffers from the large perceived cost of the local file system. If the designers had known how cheap the local file system was, they would have chosen a different implementation strategy. (Transarc is interested in using the Faster AFS concept in a later AFS release.)

No one wants to pay for transparency, we want it for free. AFS clients store files in the local file system, so we expect AFS to read data at local file system rates. We want to reach the data in the cache at no additional cost. We rationalize that AFS ought to step out of our way when we want the data in the cache.

The benefit of the hinting mechanism comes from the interaction of AFS and the local file system. With typical chunk sizes of 64K, and typical user read requests of 4K and 8K, it seems natural to provide a short cut to reach the chunk.

The direct impact of these modifications to application level programs is unclear. Typical reads by applications on the IBM RT, and the IBM RS/6000 now run significantly faster. However, it is easy to build micro-benchmarks to show insignificant

improvement. Additionally, most applications spend only a fraction of their time doing reads.

Acknowledgements

Thanks to Brian Renaud for helping me keep my nose in the paper, Peter Honeyman for pointing me at Edward Tufte, Mike Kazar for being Mike Kazar, Matt Blaze and David Richardson for reviewing over and over again, Lyle Seaman for spending more time then I ever expected, and Mary Jane Northrop for editing and editing and editing.

References

1. T. Hacker, *Netatalk Architecture*, Internal CITI Report, August 31, 1992.
2. T. Hanss, "University of Michigan Institutional File System," *IAIXTRA: The AIX Technical Review*, pp. 25–32, January 1992.
3. J. H. Howard, "An Overview of the Andrew File System," *USENIX Conference Proceedings*, pp. 23–26, February 1988.
4. S. R. Kleinman, "Vnodes: an Architecture for Multiple File System Types in Sun Unix," *Usenix Conference Proceedings*, Summer 1986.
5. Butler W. Lampson, "Hints for computer system design," *ACM Operating Systems Review, Special Issue*, vol. 21, no. 5, pp. 33–48, 1983.
6. M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A Fast File System for UNIX," *ACM Transactions on Computer Systems*, vol. 2, no. 5, pp. 181–197, August 1984.
7. M. Satyanarayanan, J. H. Howard, D. A. Nichols, R. N. Sidebotham, and A. Z. Spector, "The ITC Distributed File System: Principles and Design," *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, 1985.
8. Songnian Zhou, Herve Da Costa, and Alan Jay Smith, "A File System Tracing Package for Berkeley Unix," *Usenix Conference Proceedings*, pp. 407–419, Summer 1985.

Author Information

Michael T. Stolarchuk received his undergraduate degree in Computer and Communications Soiences from the University of Michigan. After discovering the wrong life in California (TSP/SPF/IMS on IBM 3033), he had the opportunity of bringing the first Vax/BSD UNIX to the University of Michigan Engineering College. After a brief stint in an Ann Arbor company, and after getting his M.S. in Computer, Information, and Control Engineering at Michigan, he returned to work at the University. He later joined the IFS Project at CITI. Recent projects include hijacking authentication for AFS RPC's, and AFS-to-your-protocol-here translators. His current interests are AFS translators and security, fast file systems, fast foods and fast cars. He can be contacted at mts@umich.edu, or via U. S. Mail at IFS Project, CITI, 519 W. William, Ann Arbor, Mi 48103.

The AutoCacher: A File Cache Which Operates at the NFS Level

Ronald G. Minnich – Supercomputing Research Center

ABSTRACT

The AutoCacher is a caching file system. Its most common use is to cache read-only files from remote NFS file systems to a local disk, although it can, in general, cache from any file system to any other. It is intended to provide the same type of file caching provided by, e.g., the Andrew File System.

The autocacher operates as an NFS server, not as might be expected as a Virtual File System, as do other caching file systems such as TFS[8][1] or the system described in [7]. In so doing it demonstrates that activities such as file caching, which one might expect to be required to operate at the level of a Virtual File System, can operate quite effectively at the level of NFS, despite its stateless nature.

The autocacher has been in operation at SRC since December, 1989.

Introduction

The AutoCacher is a caching file system. Its most common use is to cache read-only files from remote NFS file systems to a local disk, although it can, in general, cache from any file system to any other.

The autocacher runs as a user-level process and provides its services via partial emulation of NFS[5], in much the same way as the automounter[1] or AMD[4] do. These two automounters support emulation of a directory structure and soft links. The soft link emulation allows these programs to detect references to remote file systems; mount the remote file systems if needed; and then redirect the reference to the remote file systems.

The autocacher extends the directory and soft link emulation. The directory structure emulation supports full trees, rather than the one-level deep trees of the automounters. The trees are a shadow of the trees of the remote file system. The trees are built incrementally, as parts of the remote file system are referenced; over time, parts of the shadow trees that are not referenced are pruned, and recreated on demand.

The emulation of files is extended to emulate regular files or soft links. The regular file emulation is needed (as explained below) the first time a file is cached from remote to local disk. The soft link emulation is used for redirecting references to the remote or local version of the file. The remote reference is needed in those cases that a file can not be cached.

Certain operating system operations (such as examining files, reading files, and so on) cause NFS LOOKUP operations. When a LOOKUP operation is received by the autocacher for a file, it can resolve the operation in one of three ways:

- If there is a local copy of the file, then the link will resolve to the local copy after it is examined and passes certain tests. The local copy is examined to determine if it is obsolete. If it is obsolete, then it is ignored and the autocacher behaves as though there were no local copy of the file.
- If there is no local copy, then the remote copy is examined. If the remote copy exists, and there is room on the local file system for the file, then the autocacher responds in such a way that any read calls will be sent to the autocacher. When the autocacher is asked to perform a read, it will copy the remote file to the local cache and service the read from the local copy of the file; further LOOKUP requests for the file will cause the autocacher to return a soft link to the local copy. Note that when the autocacher supports READ operations it is actually providing the READ service that would be performed by an NFS server.
- It may not be possible to cache the file locally, as there may not be enough space on the local disk. Also, each mount point can have specified a minimum number of free kilobytes that must remain on the disk when a file is cached for that mount point. In this way, relative priorities of different remote file system caching may be established. If it is not possible to cache the file locally, then the autocacher returns a soft link to the remote file. Note the advantage this has over systems which must cache. The failover case in the autocacher is to hand the work off to NFS.

Thus, files in the autocacher are three-valued, depending on the autocacher's ability to cache them:

[1] TFS can be thought of as a cache-on-write file system

they can appear to be links to a local file; links to a remote file; or they can appear to be a regular file, and READ operations on the file are handled by the autocacher.

The autocacher was written because we found that we were making very ineffective use of the local disks on our workstations. At the time (1989) SRC ran workstations with local disks arranged in the traditional way: a / partition; a /usr partition; and a swap area. We found that at any given time only a very small percentage of the files in / and /usr were being accessed in the previous seven days – 3%. The sample was taken over approximately 100 workstations. Even on those workstations that had very active programmers using them, only 6% of the files had been accessed in the previous week.

Strangely enough this situation has only gotten worse in sites we have examined in the past few years, as the local disks have gotten much larger, currently reaching sizes of 200 and 400 Mbytes. System administrators seem to be at a loss as to what to do with all those bytes. We have seen some very strange cases:

- the entire 200 Mbyte disk is used for swap, very peculiar on a machine with only 32 Mbytes of memory
- most of the disk is left unused, with a regular root and swap partitions and an unused left-over partition
- (pathological case) All of /usr/share was put on the local disk

We decided to put our unused space to better use. We deleted all the unused trees in / and /usr and remote-mounted them. We then cached into the freed-up space.

The original version of the autocacher was derived from the SunOS automounter. In fact, since the automounter implements a limited user-level NFS service, it was a very good starting point.

One other advantage of the autocacher is that once a program is run from the cached local copy, that program will have access to its text until it exits. Thus we have eliminated the familiar NFS problem of a file being replaced and causing processes paging text from that file to crash. Since the program is being paged from local disk, the standard mechanisms apply for ensuring that the program has access to the file until the file is no longer needed.

Related Work

The idea of moving files from high-latency to low-latency storage when they are accessed is an old idea. For decades mainframes have had systems that move files from tape to disk when they are accessed, and back again when they are done. Andrew demonstrated a caching file system for use on networks with file servers and workstations[2]. The Andrew file system initially ran only at CMU, but is now distributed by TransArc. More recent work concerns user-level file cache management on Suns via the vnode interface[7]. The IEEE Mass Storage Reference Model envisions migration in some form for efficiency[3]. The Coda[6] file system supports both file migration and disconnected operation, in which the server need not even be up for file system operations to function at the client. Unitree also supports file caching.

All of the systems mentioned above require kernel recompilation and extensions in order to be used (and, in the case of Andrew, purchase orders, which can be even more difficult). By contrast, the autocacher requires no kernel changes whatsoever, and can be started up just as any automounter is. The autocacher is designed to support file caching from read-only file systems. The design is such that the server can be down for long periods of time and the autocacher will still work.

Description of the Autocacher

In the following discussion, we will be using the terms remote, local, and emulated. By remote, we mean the file system from which we obtain files. By local, we mean the file system on which we are storing cached files. By emulated, we mean that NFS operations will be handled by the autocacher directly. READs for local and remote files are handled by the local UFS and the remote NFS servers respectively. Redirection of file system requests for those files to these other servers is accomplished by soft link emulation, as in the standard automounter. READS for emulated files are handled by the autocacher.

```
# Emulate three directories in /cache/test:
#  test, local, and public. The remote directories
#  are on the left side of the :,
#  the cache directories on the right
#  No options specified here.
/cache/test          /home/rminnich/test:/var/cache/test
/cache/local         /usr/local/bin:/var/cache/local
/cache/public        /net/public.bin:/var/cache/public
```

Figure 1: A Sample Autocacher Configuration Table

out-of-date by checking the modification time, size, and other attributes of the remote copy. Note that this mechanism can be spoofed by a sufficiently determined person, but this spoofing is not an issue for read-only NFS file systems – not at SRC, anyway. The file is copied if the cached version has been accessed within the last day. Otherwise, the autocacher behaves as though there is no cached copy, and waits for a READ request to initiate bringing the new copy over.

- The file is not in the cache. In this case, the autocacher must determine whether the file is being opened or just examined. Blindly copying files when they were examined would result in a very ineffective cache. This is the most interesting part of the protocol; we will discuss it below.

Determining whether to copy a file

As described above, the autocacher may find that a local copy does not exist, and it must determine whether a file is actually going to be read. When the autocacher receives a GETATTR request, in this case, it returns information for a regular file with a stat buffer copied from the remote file. Thus the next NFS READ (if there is one) is directed to the autocacher. When the autocacher receives this READ request, it copies the file to the local cache directory. If the file is small, then pages are allocated and the entire file is read into memory. If the file is large, it is mapped into memory. Thus READ requests are resolved by a simple pointer computation, rather than actual read system calls. This results in good performance on READ requests.

Thus the files are emulated via a three-state structure. The structure changes state as driven by the state of the cache; the state of the structure itself; and requests from NFS clients.

Handling directories

The autocacher provides full support for directory trees. When it reads the configuration file and determines the name of the remote directory, the autocacher scans the remote directory. Any directories encountered are treated much as files, save that their internal representation has a tag attached that indicates that it is a directory that is not yet scanned. The directory will not be read until an NFS READDIR request is received for that directory. Thus the directory is only read on demand. This behavior saves a substantial amount of both space (for the internal representations of all the files) and time (it could take quite a while to walk down some of the remote trees), while providing full access to any subdirectories specified in the configuration file. We needed to add two new node types for this change; a type called NF_UDIR, for unread directories; and a type called NF_DIR, for read-in directories.

Directories may be accessed and then not used for a long time. The autocacher will prune the internal representation of the directory tree and eliminate parts of it which are not accessed for over a few hours. This saves virtual memory. Thus, a directory can make the transition from unread to read and back again.

Determining whether a file is obsolete

While we could simply stat the remote file each time a name needs to be resolved, in practice we do not do this. We wish to take advantage of the fact that the automounter is also active on this system; the file systems are being unmounted, making the workstation less vulnerable to temporary server outages. We therefore only check a file's obsolescence if two conditions are met:

- it has not been checked in more than an hour
- the time since the autocacher was started, modulo 60 minutes, is between 55 and 59 minutes.

The reason for the first condition is to allow the automounter to unmount file systems. The reason for the second is more complex: if we simply did the stat of files as they got to be over an hour old, the remote file systems would always be mounted, as at any given time just about any file we access would be over an hour old (experience showed this to be true). We initially moved to checking files that were an hour old, but only in a window defined as the last five minutes in the hour. We have found in another context an absolute window of this type can lead to 'automounter storms' in which many automounters make mount requests at once, leading to some of the mount requests failing. Until this problem is fixed[3], our window is relative to the autocacher startup time, in an attempt to spread out the automounter requests. Note that if a lot of workstations are booted at once, this time-relative window will fail in the same way the absolute window does.

The file time limit can be set via a mount option in the configuration file; we have co-opted the *timeo* option for this purpose.

For the reference, we can use the remote file's modification time. For the local copy we also use the file's modification time; we change it to reflect the last time we checked it. There remains the problem of determining when a local cache file was actually created, and for that we use the ctime. The very last operation we perform after a file is copied is a chmod, which sets the ctime to the time we consider as the creation of the file.

If a local file is determined to be obsolete, it is ignored. A new one is not copied over unless the atime indicates that it has been accessed within the previous 24 hours or at the next READ request.

[3]The problem, reduced to its essentials, is that SunRPC uses UDP.

One case that can be difficult to handle is when the remote file is deleted. It is not always possible (i.e., the return error does not tell you enough) to differentiate between a file being unavailable (due to server outage) and a file no longer existing. We adopt the convention that if the remote file cannot be stat'ed at all, then the local copy is ignored if it has not been accessed or modified in more than seven days. The user then sees the file as no longer being available. The reason for the seven day window is described below.

Actual removal of files is handled by a separate process, invoked only when the local disk becomes too full.

Determining whether a file is unused

We define unused files as those that have not been accessed for more than seven days. The autocacher does not always manage these files, since they may not be accessed even to be examined. We currently use a standard find script to delete such files. If the disk is less than 80% full, we do not delete anything. If the disk is more than 80% full, then all files that have not been accessed or modified in over seven days are deleted. That time is also adjustable but we have found seven days to be the best limit for our purposes.

There is a complication, caused by NFS statelessness, that requires us to be careful about closing the seven day window to some smaller value. It is not inconceivable that a long-running program could page fault on text after a long inactive period and present a LOOKUP request to the autocacher. If the autocacher had previously been satisfying reads to this file, it is imperative that this lookup resolve to the cached file, as that file represents the program image being executed. The remote NFS copy might have changed in the meantime, so that remote copy can not be used. Sometime during the seven days, there is a very good chance that a programs use of a file will be preceded by a LOOKUP/GETTATTR cycle, which will in turn direct it to the local cached copy. Once the program is paging text from the cached copy further changes in the remote version will not affect it, as the cached copy is retained (by UFS) as long as there are users of it. This problem is fairly hard to exercise in practice, and occurs only in the rare event that a LOOKUP occurs a very long time after a set of READs have been handled by the autocacher. This timeout value in practice has not failed in two and one-half years.

We are experimenting with what to do if the disk is still over 80% full and all old files have been deleted. In practice this has not happened.

Sample Runs

In order to better show the operation of the autocacher we are going to work through a sample run of a program accessed via the autocacher. The program is known to the user as /usr/local/bin/sun4/inc. The local directory is /usr/local, with several autocacher mount points, one of them being bin. The autocacher is caching bin from /net/bin, and is caching into /var/cache/bin. Thus, as the kernel works through the path, it will eventually get to the bin part of the name, as shown in Figure 3.

The autocacher stats the remote file, and builds a node of type UDIR in the in-memory representation of the file system.

In the next figure, Figure 2, the kernel has executed a READDIR on bin, followed by a LOOKUP on sun4. The autocacher executes a readdir for

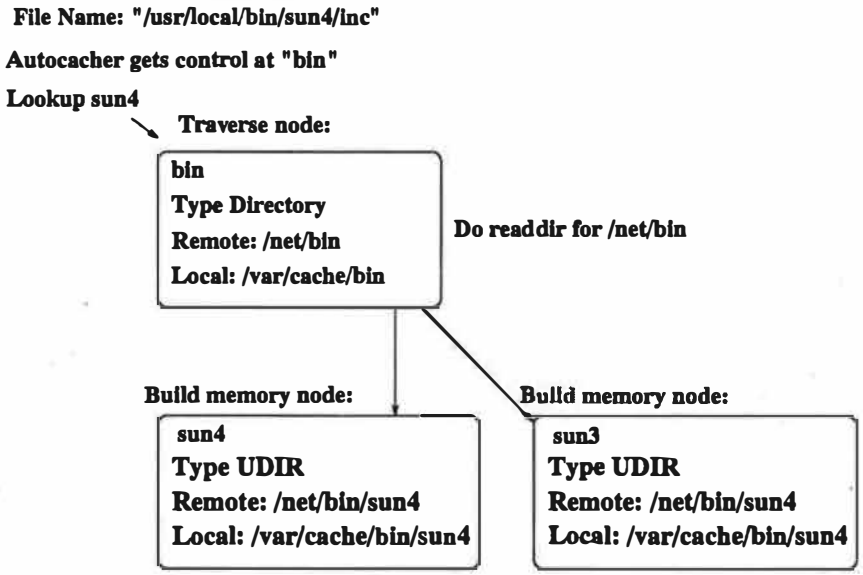


Figure 2: Filling in information for the bin directory

/net/bin, and builds UDIR nodes for the sun3 and sun4 directories. The node type for bin changes to Directory.

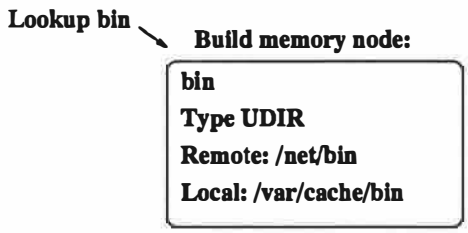


Figure 3: Building a path starting with bin

Finally the kernel requests a READDIR for the sun4 directory, followed by a LOOKUP on the file named inc, with the results shown in Figure 4. The autocacher finds a local copy of inc, so the mode is set to local-soft. The kernel may do a READLINK operation on the file, in which case it will be returned the value /var/cache/bin/sun4/inc, and will then get the real file from the local disk.

We now show various scenarios for resolving cache entries. In Figure 5, the local file is useable, so the type of the autocacher node resolves to a soft link to it.

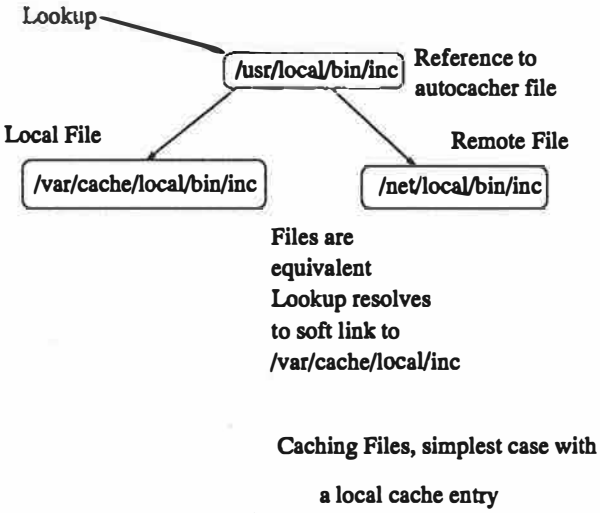


Figure 5: Resolving to a local cached copy

In Figure 6, there is no local cached copy. In this case, the autocacher will return a file handle for a regular file, and field the READ calls itself. The next LOOKUP will change the node type to resolve to a local soft link.

In the next example (Figure 7), there is no cached file and there is no room to cache one. Therefore the LOOKUP resolves to a link to the remote file.

Future Work

The autocacher is currently stable and tested on a number of workstations, both Sun 3 and Sun 4. It has been implemented via modification to both AMD

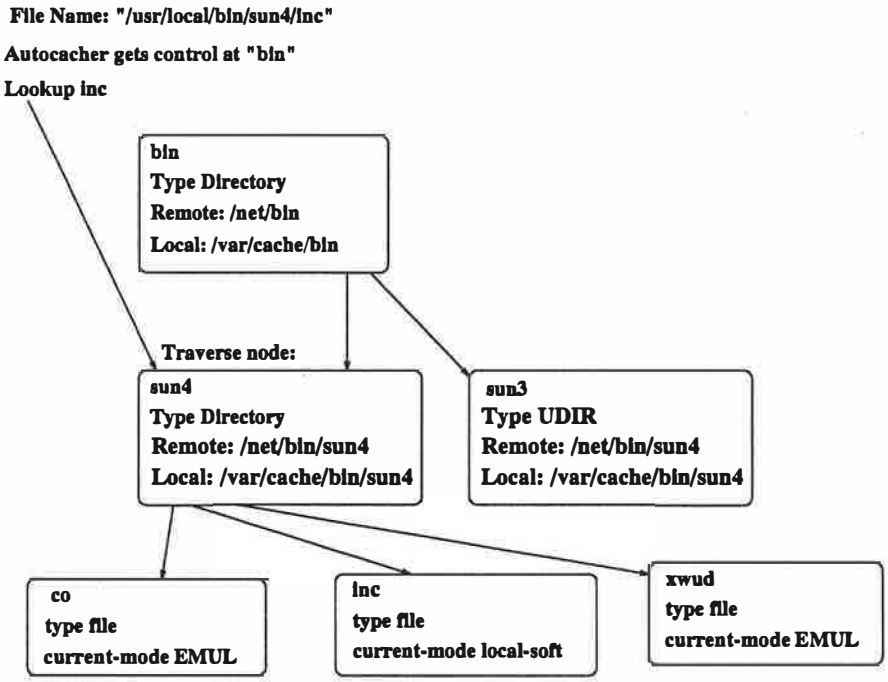


Figure 4: Filling in information for the sun4 directory

and the automounter. While we initially liked AMD better, the complexity of AMD has us leaning back to using the automounter-based version.

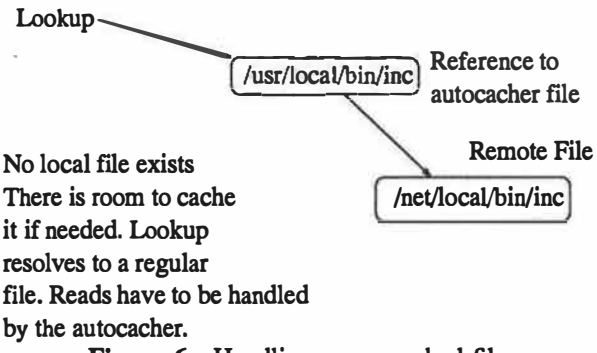


Figure 6: Handling a non-cached file

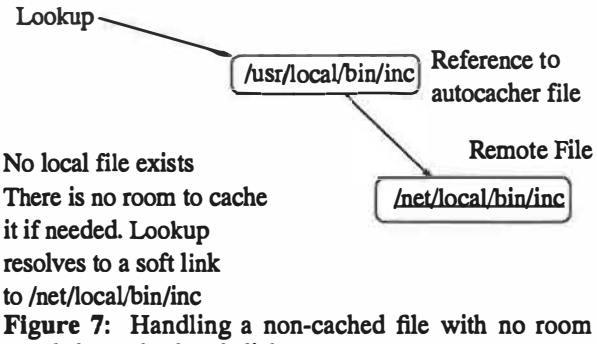


Figure 7: Handling a non-cached file with no room left on the local disk

We may yet tackle the task of caching writeable files as well. This is a considerably more difficult job, as our current version which caches read-only files is, by definition, insensitive to system outages. If the copy operation fails at any point, the local file will have a different size than the remote reference file, and will be replaced. The key idea here is that remote operations (via soft link to a remote file) may be substituted for local operations (via soft link to a local file) with no effect on the user. For the writeable case, this equivalence does not exist.

The hardest part of caching writeable files is not technical. Were we to cache writeable files we would be instantly accountable for every lost file in the building, even if we could get users to trust us that long. The effort of getting such a system into common use is not a small one.

Summary

The autocacher is a caching file system which operations at the NFS level, demonstrating that file caching need not be implemented at the VFS level. It has been in operation at SRC since December, 1989. It was written so that we could make more effective use of the local disks on our workstations. To our surprise, over the last few years, the utilization of local disks on workstation has gotten less efficient as the disks got larger. Thus the case for an autocacher has grown stronger.

Because the local disks are not backed up, we only cache files accessed as read-only. We do not attempt to cache writable files. We may consider doing such caching in the future. A key problem is not technical: convincing people to trust new systems that do things differently is always very difficult.

Bibliography

[1] Brent Callaghan and Tom Lyon. The automounter. In *Usenix Winter 89*. Usenix, 1989.

[2] J.H. Howard, M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, and M.J. West. Scale and performance in a distributed file system. *Communications of the ACM* , 6(1):51-81, February 1988.

[3] IEEE. Mass storage system reference model: Version 4 (may, 1990). Technical report, IEEE, May 1990.

[4] Jan-Simon Pendry. Amd – an automounter. Technical report, Department of Computing,Imperial College, 1989.

[5] R. Sandberg, D. Goldberg, D. Kleiman, S. Walsh, and B. Lyon. Design and implementation of the sun network file system. In *Usenix Summer 1985*. Usenix, 1985.

[6] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, and E. Siegel. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. Computing* , 39(4), April 1990.

[7] David C. Steere, James J. Kistler, and M. Satyanarayanan. Efficient user-level file cache management on the sun vnode interface. Technical report, Carnegie Mellon University, April 1990.

[8] Sun Microsystems. Translucent file system. In *SunOS Reference Manual*. Sun Microsystems, 1990.

Author Information

Ron Minnich is a Research Staff Member at the Supercomputing Research Center, Bowie, Md. His interests include Operating Systems support for distributed computation, high-bandwidth networking, and special-purpose architectures. Other recent work includes the Mether distributed shared memory and debuggers for the Splash system, a programmable linear array based on Xilinx 3090 chips.

Pitfalls in Multithreading SVR4 STREAMS and Other Weightless Processes

Sunil Saxena, J. Kent Peacock, Fred Yang, Vijaya Verma, Mohan Krishnan – Intel Multiprocessor Consortium

ABSTRACT

As part of the effort of creating a multiprocessor version of System V Release 4, the Intel Multiprocessor Consortium attempted to multithreaded the kernel STREAMS subsystem. STREAMS are a System V facility which provides a messaged-based communications framework, primarily for use in providing pipe-like configurability for character devices. Multithreading STREAMS required a significant amount of work and it was quite difficult to achieve a "correct" solution. In fact, three different versions of the locking were necessary to solve a number of significant problems which were identified. As such, this effort represents an interesting case study for the type of difficulties encountered in multithreading a complex subsystem. In particular, a very fine-grained multithreading strategy was tried first and found to have undesirable deadlock, performance and stability characteristics. Subsequent versions allowed less apparent parallelism, but actually improved all of these properties.

The root cause of many of the problems encountered was the existence of "weightless processes", that is, control threads which do not have their own processor stacks. Examples include interrupts, timeouts and STREAMS processing. The major drawback to weightless processes is their inability to suspend execution to wait for an event or resource, thus making them susceptible to deadlock. A number of examples of weightless process deadlocks are explored to illustrate the disadvantages of this approach, particularly in a multiprocessor system.

Introduction

The performance of STREAMS is an important factor in determining the capacity of servers in UNIX networks, as both the ASCII terminal and network communications protocols are implemented using STREAMS in System V Release 4. As servers evolve to the point where multiprocessors are the rule, rather than the exception, scalability of multithreaded STREAMS and STREAMS-based network protocols will assume great importance. With this in mind, much effort was put into the STREAMS multithreading effort undertaken as part of the Intel Multiprocessor Consortium's SVR4MP release.

What are STREAMS?

STREAMS are a support framework provided in UNIX to enable a high degree of configurability and modularity in communications drivers. They are a kind of "pipes-in-the-kernel", with a few differences from user-level UNIX pipes. The main differences are that STREAMS are bi-directional and message-based. In fact, in SVR4, pipes are implemented using the STREAMS framework.

As a data abstraction, a STREAM is fundamentally a set of queue pairs, one for each direction, connected together in a linear list. Each queue pair is bound to a module which implements some function, such as terminal character processing. Modules are essentially the "filters" of these kernel-pipes, and can be dynamically added and removed from each stream. Each queue in a pair has two message processing functions associated with it: the put and service routines. The put routine is called with a queue pointer and a message, and does some or all of the processing required on input of a message to that particular module. If the put routine cannot completely process the message, it can be put onto the actual queue and the service routine enabled to run. Services routines are executed by a STREAMS dispatcher at selected points within the kernel, such as between processor dispatches and before returns from system calls. As such, STREAMS are similar to interrupts and timeouts in that they do not own their own contexts, but borrow the current process' stack. More details about the structure of STREAMS can be found in Ritchie's original paper [Ritchie84], or in brief summary form in a Sequent paper [Garg90].

STREAMS Locking

Most multithreaded STREAMS approaches try to achieve the maximum available concurrency [Garg90, Barton92]. In practice, this means that the generic locking does not preclude put and service routines for a given queue from running in parallel with each other or with all others in the system.

Indeed, the Consortium's original locking strategy had precisely this goal, and the initial implementation was designed to achieve it.

There are a number of main areas in STREAMS where races can occur and locking is required. The obvious locking requirement is the protection of the queues themselves while messages are added and removed. STREAMS configurability allows modules to be pushed and popped dynamically, and closing a stream results in the disassembly of the entire stream. Such so-called plumbing operations must be synchronized with queue activities to delay them while put or service routines are active. In addition to the locking of the actual queues, the internal data maintained by the module associated with each queue must be protected. Lastly, the uniprocessor model of STREAMS assumes that only one system call can be active at once. It is usually important to preserve these semantics to avoid drastic re-writes of existing modules, and to simplify synchronization within the STREAMS subsystem itself.

Version I

The original version of the streams locking attempted to provide the most concurrency possible, by using very fine-grained locking. It was largely inherited from NCR's original SVR4 multiprocessor effort [NCR91] and involved the use of the following locks:

- A lock on each queue structure, held during queue manipulations.
- A lock on the stream, to protect manipulations of queue reference counts and plumbing state information. These counts are used to avoid the races between plumbing and queue operations.
- A lock on the stream head data.
- Miscellaneous locks to protect scheduling queues and various free lists.
- Vnode or fifo locks above the stream head to serialize system calls.
- Locks maintained by each module for its internal data.

One of the first things discovered about the use of a separate lock for each queue was that queue interactions are highly circular. The most common method of passing a message from one module to the next is the *putnext* function, which calls the put routine of the next queue, passing it a message. A reply to a message can be sent back on the opposite-direction queue using the *qreply* primitive, which also calls the appropriate put procedure. Examples of qreply use include echoing characters and responding to *ioctl* messages. Using only these two primitives, circular queue dependencies can be made to occur quite easily. As a result, holding a lock on one queue while performing an operation on the next queue cannot be done without causing deadlocks.

Most implementations avoid this type of deadlock by requiring that all locks be released across operations that involve adjacent queues. This does not involve only the queue locks, but also the locks used by a module to protect data which are private to a queue. This was the approach adopted initially by the Consortium. There were situations discovered where this was not possible, so one of the first concessions to the ideal of maximal concurrency was to combine all of the queue locks for a given stream into a single stream lock. Since the Consortium locks allow recursion [Intel92], obtaining the stream lock multiple times to protect multiple queues avoided these deadlocks. On the other hand, most of the time the lock was released and reacquired across adjacent queue operations.

There are a number of significant problems with this approach. Firstly, there were simply too many lock and unlock operations performed. A test of the networking code showed a total of 80 lock operations to process a single TCP/IP packet on a local network. The second problem was that serialization of operations was lost due to the fact that queues and local data were unlocked. This resulted in possible out-of-sequence data – not acceptable behavior for a TCP connection. Most of the lock and unlock operations were used to protect the plumbing synchronization information, namely the queue reference counts and plumbing state. In addition, large degradation relative to the original uniprocessor code was observed and no benefit was achieved by adding additional processors. The locking was simply too fine-grained, with the time to obtain and release the locks dominating the actual protected code. It is worthwhile to study the details of the plumbing locks to see how such a large number of lock operations is required.

Plumbing Locks

One of the main synchronization issues in multithreaded STREAMS is the prevention of changes to the structure, or plumbing, of a stream while module routines are running or while there are still active references to a queue. The state used to achieve this synchronization includes two additional fields in each queue structure, *q_plumbing* and *q_ref*. The *q_ref* field is a reference counter which counts the number of active service or put procedures on the queue. The *q_plumbing* field contains several flag bits, the most important of which are QPROCSON and QWANTDET. The QPROCSON flag must be set for any put or service routine to be allowed to run. The QWANTDET flag indicates that a process wants to detach the queue or otherwise change the plumbing.

An abstracted outline of the relevant functions is shown in Figure 1. The actual functions must deal with a pair of queues, one for each direction of data travel, whereas the code shown illustrates only one queue. The functions are fairly standard UNIX

paradigms, including the use of *sleep/wakeup* event signaling. This is possible due to the Consortium's locking semantics, which include the release and reacquisition of locks across context switches. The locking primitives are described in detail elsewhere [NCR91, Intel92].

Examination of this code shows that each put or service procedure call involves acquiring the stream lock twice. The module procedure must in all likelihood also acquire a lock to protect its private data, since concurrent put or service processing is allowed. This yields a total of at least 3 locking operations per primitive stream operation, and gives some insight into why the locking overhead is so large.

Version II

To solve the primary performance problem, a second version of the locking was implemented in which the goal was to reduce the number of locks required to protect the plumbing information. This was done by replacing the reference count and plumbing information locking with very lightweight

```
/* Enable put and service processing */
qprocson(q) {
    STREAM_LOCK(q);
    q->q_plumbing |= QPROCSON;
    STREAM_UNLOCK(q);
}

/* Wait for and disable put and service processing */
qprocsoff(q)
{
    STREAM_LOCK(q);
    qrefwait(q);
    q->q_plumbing &= ~QPROCSON;
    STREAM_UNLOCK(q);
}

/* Wait for running put and service routines */
qrefwait(q)
{
    /* Called with stream lock held */
    while (q->q_ref) {
        q->q_plumbing |= QWANTDET;
        sleep(&q->q_ref, PZERO);
    }
}
```

In put and service dispatch routines:

```
    STREAM_LOCK(q);

    check q_plumbing & QPROCSON and skip over queue if not enabled
    q->q_ref++;

    STREAM_UNLOCK(q);

    call put/srv procedure

    STREAM_LOCK(q);

    if ((--q->q_ref == 0) && (q->q_plumbing & QWANTDET)) {
        q->q_plumbing &= ~QWANTDET;
        wakeup(&q->q_ref);
        return;
    }

    STREAM_UNLOCK(q);
```

Figure 1: Version I Plumbing Synchronization

atomic operations. With this change, the number of stream locks was reduced by 50%, but the performance only improved by 25%. In addition, this change did not solve the scalability or correctness problems previously discovered.

Figure 2 shows the code which uses the simple atomic operations to replace the functions shown in Figure 1. These atomic operations are defined so that `atomic_op(arg1, arg2)` performs the operation `arg1 op= arg2` atomically and returns the old value of arg1. The semantics of the synchronization require that tests of the QPROCSON flag and manipulations of the reference count be atomic. Combining the q_plumbing and q_ref fields almost allows the original semantics to be maintained, since the reference count can be incremented and the QPROCSON flag tested atomically. There is a window in qrefwait, however, where QPROCSON can be turned off briefly while a put or service routine is trying to increment the reference count and test QPROCSON, resulting in a message being dropped when the queue has not truly been disabled. Although this version reduced the locking overhead substantially by replacing stream locks with a simpler locking protocol, the overhead was still unacceptably high. Thus, Version III locking was developed.

Version III

The solution to all of the previous problems was to retreat even further from the maximal-concurrency model. This was accomplished by holding the stream lock throughout all system calls, interrupt processing, put routines and service procedures. In effect, there is only one operation per stream allowed at a time. This model still maintains parallelism among independent streams, however. In addition to the single stream lock, a simple spin lock was added to each queue, which allows addition or removal of messages without holding the stream lock. This is quite important in avoiding deadlock in multiplexor configurations, which are discussed in a later section.

The final version of the synchronization primitives is shown in Figure 3, where all of the explicit locking has vanished, because the stream lock is held around each entire operation on a stream.

```
/* Enable put and service processing */
qprocson(q)
{
    atomic_or(q->q_ref, QPROCSON);
}

/* Wait for and disable put and service processing */
qprocsoff(q)
{
    qrefwait(q);
}

/* Wait for running put and service routines */
qrefwait(q)
{
    /* QREF selects reference count bits */
    while (atomic_and(q->q_ref, ~QPROCSON) & QREF) /* while count != 0 */
        atomic_or(q->q_ref, QPROCSON);
}
```

In put and service dispatch routines:

```
    if (((atomic_add(q->q_ref,1)) & QPROCSON) == 0) {
        /* Queue is disabled */
        atomic_add(q->q_ref,-1);
        freemsg(mp); /* Drop the message */
        return;
    }

    call put/srv procedure

    atomic_add(q->q_ref, -1);
```

Figure 2: Version II Plumbing Synchronization

Where's the Work Done?

In working with and multithreading STREAMS, an impression has developed that is contrary to an opinion expressed by Garg [Garg90], namely that most of the processing of STREAMS is done in the service routines. In the sense that STREAMS processing can also be done from system call entry points, timers and interrupt routines, this characterization is probably true relative to these other processing modes. However, if the meaning of the statement is that most inter-module message-passing involves scheduling the adjacent service routine, this does not seem to be the case. Rather, kernel debugger stack traces have shown that putnext functions from one module to another tend to frequently nest quite deeply. In addition, reading the code reveals a preponderance of putnext over *putq* calls. (The putq function takes a message, puts it onto a queue and enables the queue's service procedure, whereas the putnext and qreply functions call the queue's put procedure, passing the message and queue pointers.) In fact, static counts over the entire kernel source revealed a total of 109 putq calls, 530 putnext calls and 348 qreply calls. These counts imply a ratio of 8:1 put routine versus service routine processing, at least for the initial handling of a message. Counts of the static configuration of modules showed that of 99 STREAMS modules examined, 76 contained put routines, while only 51 had service procedures. With a kernel test instrumented to count calls to put and service routines, sending 10,000 packets through a loopback TCP/IP connection resulted in 73,184 put routine calls and 45,311 service routine calls, for a ratio of 1.6:1.

The fact that there are an average of 4.5 service routines run per packet suggests that allowing service routines to run in parallel might yield a maximum speedup of around 4. However, the TCP/IP connection uses 3 streams, so that locking at stream granularity allows a maximum speedup of 3. For this example, the additional locking overhead in the maximum-concurrency approach does not justify the increase in potential parallelism. The maximum possible speedup is unlikely to be achieved in any case, because there is likely to be one service routine that takes more time per packet than the others which will limit the maximum throughput.

This analysis suggests that a strategy which attempts to optimize the simultaneous running of adjacent service procedures will produce less parallelism than might be expected. There is substantial overhead in scheduling processing to happen in service procedures due to the context switching involved. Although having sets of processors running service routines in parallel on a single stream has intuitive appeal, in practice it appears that the processing at each module is too small to justify the overhead in cases examined so far. Hence, allowing put procedures to hold the stream lock for the entire duration of the operations up and down the stream reduces the overhead dramatically, without any clear sacrifice in real scalability.

These changes resulted in significant performance improvement in the TCP/IP test described. Addition of a second processor increased the throughput of the test by 78% using the best available hardware. The stream lock was acquired a total of 20 times per packet, down from 80. However, of those 20, 15 were recursive acquisitions, which are quite cheap. Queue lock acquisitions totaled 37 per packet.

```
/* Enable put and service processing */
qprocson(q)
{
    q->q_plumbing |= QPROCSON;
}

/* Wait for and disable put and service processing */
qprocsoff(q)
{
    q->q_plumbing &= ~QPROCSON;
}
```

In put or service dispatch routines:

```
    if ((q->q_plumbing & QPROCSON) == 0) {
        freemsg(mp); /* Drop the message */
        return;
    }

    call put/srv procedure
```

Figure 3: Version III Plumbing Synchronization

A benefit of this model is that a large degree of locking protection is inherently provided to the modules comprising a stream. Much module code which had been multithreaded was changed back to its original state by the removal of locking code. Using the stream lock to protect the module's local data, as well as the streams data structures, proved to be quite effective. In object-oriented terms, one can think of this as a kind of "lock inheritance", where the modules use the availability of the stream lock. In addition, the state of the stream remains intact through entire spans of processing, which is much closer to the uniprocessor behavior. This eliminates the problem of out-of-sequence data as well. The number of modules on a given stream is likely to be small, whereas the number of streams can be logically unbounded. This argues for expending multithreading effort to make sure that parallel streams scale well, and worrying less about parallelism within a stream.

Multiplexors

The Version III model finally used to lock individual streams was quite satisfactory but for one complication: multiplexors. Multiplexors are a type of STREAMS module which allow the connection of unrelated streams to one another, as in network protocols such as TCP/IP. The TCP module multiplexes a set of connections onto a single stream connected to the IP module, which in turn multiplexes different network protocols, such as TCP and UDP, onto (possibly multiple) streams which connect to network hardware devices. The main complication with multiplexors is that the logical flow of control crosses stream boundaries, such that one can be in the context of a TCP connection stream and require access to the IP stream, or vice versa. This situation represents a circular dependency which is sufficient to allow deadlocks to occur. When this problem was first noticed, it looked as if the only solution might be to have a single lock for all of STREAMS, due to the tangled nature of some of the interactions!

One solution to this problem, which is specified by the multiprocessor DDI/DKI from Unix System Laboratories [USL91], is to not allow put procedures of one side of a multiplexor to be called from another. Unfortunately, this solution is only practical if one is willing to expend the effort to restructure multiplexor modules, such as the existing SVR4 TCP and IP modules to satisfy this constraint. The Consortium solution to this problem involved the definition of a STREAMS module which was called a *coupler*. The coupler was designed to be inserted just below a multiplexor in a stream. At this point, it is possible to conditionally attempt to acquire the stream lock for the stream on the other side of the multiplexor before calling its put routine. If the lock acquisition fails, the coupler queues the message on its queue and tries later from its service procedure.

Figure 4 shows the simplified code structure for the write side of the coupler module, which is the side that backs off in potential deadlock situations.

```
/* Streams coupler write put routine */
strc_wput(q, mp)
{
    /* Crossed a stream boundary - stream lock is not held */
    if (TRY_STREAM_LOCK(q) != FAIL) {
        /* Empty queue if anything there */
        while (tmp = getq(q))
            putnext(q, tmp);

        putnext(q, mp);
        STREAM_UNLOCK(q);
    } else {
        /* Can't get lock - queue message */
        putq(q, mp);
    }
}

/* Streams coupler service routine */
strc_wsrv(q)
{
    /* Stream lock is held */
    while ((mp = getq(q)) != NULL)
        putnext(q, mp);
}
```

Figure 4: Streams Coupler Module Logic

The coupler module would normally reside beneath the stream head of the lower stream of a multiplexor module. The strc_wput routine would be called from the upper side of the multiplexor with the lock for the upper stream held, but not the lock for the lower stream. If the attempt to acquire the lock succeeds, any messages on the queue are removed and sent to the next module, followed by the message passed in as *mp*. If the lock attempt fails, the putq routine must be able to put *mp* onto the coupler's queue without holding the stream lock. This was a primary motivation for retaining a separate lock for each queue to protect queueing operations. When the service procedure runs, it is called with the stream lock held, and simply removes messages from its queue and passes them to the next queue. The queue locks are used to protect primitive queueing operations and some manipulations of fields in the queue structure. As such, it was possible to implement the queue locks using the most primitive form of simple spin locks, and hence they are quite efficient.

Performance Measurements

The performance results previously stated were obtained using a very simple testing strategy. The most important application of STREAMS from a practical point of view is the implementation of networking protocols, particularly TCP, UDP and IP. Larger multiprocessor machines are likely to see their greatest application as network servers, exporting processor cycles and shared file systems to clients on the network. Hence, the test chosen to characterize the effect of adding CPUs measured the packet throughput on one or more TCP/IP connections.

The test was run with varying units of load on configurations consisting of 1 or 2 cpus. The machine used to perform the multiprocessor tests was a Compaq Systempro utilizing two 33 MHz. Intel 486 processors. Each unit of load was generated by using the SVR4 *spray* command, as follows:

```
% spray localhost -l 100 -c 10000
```

This command sends 10000 packets of length 100 bytes through a TCP/IP connection which is looped back by IP. The loopback mode was chosen to avoid limitations due to network hardware, and to measure the pure CPU-bound scalability of the network protocols. For comparison purposes, results were also measured using the uniprocessor SVR4 kernel to compare degradation introduced by adding locks. The uniprocessor numbers are not directly comparable, however, since they were run on a different single-processor 486/33. In any case, the single CPU numbers are virtually identical for the SVR4MP and SVR4.0 Version 4 kernels.

Test results are summarized in Figure 5. It should be noted that the 2 CPU numbers are better even for 1 copy of spray running. Each copy of spray uses three streams, 2 TCP streams connected to each end of the TCP connection and a common IP stream between them. The table indicates that a small degree of parallelism, about 6%, is achieved with only one spray running. The results are better with 2 sprays running, yielding a 24% increase in throughput when the second CPU is enabled. Although in absolute terms this improvement appears anemic, there are some mitigating factors to be considered: The first is that the Systempro does not scale very well under load due to memory bus interference, yielding between 1.2 and 1.7 out of 2 processors in cases where software contention is known to be minimal or non-existent. Secondly, this situation is a great improvement over our original implementation, where the enabling of the second processor lowered throughput. Lastly, the implementation is much more robust in avoiding races and deadlocks that characterized the Version I implementation.

Throughput in Packets/Second				
Kernel	CPUs	Loads		
		1	2	3
SVR4MP	1	429	435	422
	2	454	541	526
SVR4	1	420	425	422

Figure 5: STREAMS Throughput Measurements

Additional Performance Results

An opportunity arose to run the spray test on a machine with 5 CPUs and better hardware scalability. In this case, each sample point was generated by running 5 copies of spray on *n* CPUs. A graph which shows increases in throughput with addition of processors is shown in Figure 6. From this graph, the ratios of throughputs relative to one CPU are 1.78 at two processors, 2.35 at three processors and 2.93 at five processors. The scalability up to three processors is quite reasonable, degrading as the fourth and fifth processors are added. This graph actually represents a lower bound on the actual scalability obtainable, the reason being that the other end of the TCP/IP connection for all of the sprays is a single spray daemon process, which is a point of contention. It is quite probable that constructing a test with a set of completely independent connections would show better scalability. On the other hand, it is also possible that the contention is due to the saturation of the common IP stream used to loop TCP packets from each spray back to the daemon process. This requires further investigation.

Weightless Processes

STREAMS are actually an instance of a type of control flow within the kernel which could be called a "weightless process" (WP). The term aptly describes processing which occurs on borrowed time

and stack space. Other examples include hardware interrupts and timeouts. The basic difficulty which arises when no stack is available is that there is no context in which to suspend the WP so that it can wait for resources or events. In particular, it is very difficult to avoid deadlock in a multiprocessor because WPs cannot sleep to wait for locks. Waiting for resources, even in the uniprocessor kernel, typically involves scheduling a call from one of a set of *ad hoc* event schedulers. For example, the following code logic is actually a paradigm inside SVR4 service procedures:

```
if (allocate a buffer == fail) {
    if (queue callback to get one
                                == fail)
        timeout(try again later);
    return;
}
```

This example illustrates two levels of resource denial, the first where the buffer cannot be allocated and the second where the structure to queue the callback cannot be obtained. If the timeout cannot be queued, the timeout function is called immediately, which turns this into a polling wait.

In SVR4MP, no solution to this issue was designed or implemented. The purpose of mentioning it here is to point out the general nature of the problem, particularly with respect to multiprocessor locking. In Solaris 2.0 [Eykholt92], a solution to this problem has been implemented, namely kernel threads, which allows interrupts and other WPs to suspend, at least for the purpose of mutual exclusion. Resource waiting is somewhat trickier, as the number of threads required to allow every WP to suspend might be unacceptably high for STREAMS. In general, every service procedure could require its own thread. Sun has also observed that using kernel threads to do interrupt processing "helps a lot" in relieving STREAMS lock hierarchy problems [Barton92].

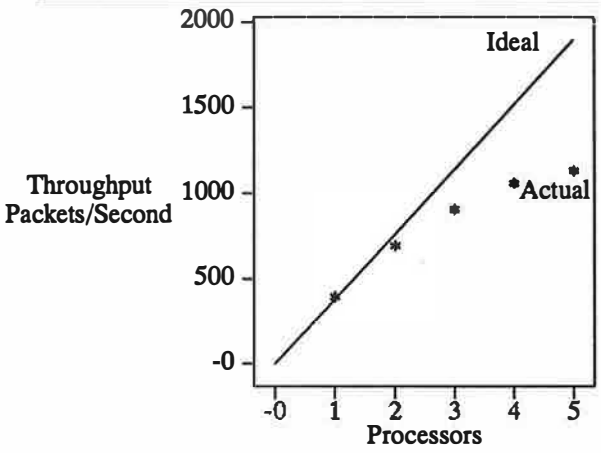


Figure 6: Scalability of *spray* test

Timeout Deadlocks

The UNIX kernel *timeout* function is a very useful facility for allowing device driver writers the ability to schedule callback functions after an arbitrary time delay. Such timeouts are typically used either to poll repeatedly for the occurrence of a relevant event or to invoke an error recovery action if an expected event does not occur (i.e., a watchdog timer). It is the latter use that can cause deadlock in a multiprocessor situation, since it exemplifies a race condition that can occur.

The race condition that leads to the deadlock happens when the *untimeout* function is used to cancel a watchdog timeout which is pending. For the deadlock to occur, a certain set of locking

```
mytimeout(arg)
{
    LOCK(mylock);     ← spins waiting for lock on CPU 0
    ...
}

myclose(dev)
{
    LOCK(mylock);

    /* Do some work */

    /* Clean up timer */
    untimeout(mytimeid); ← spins waiting for timeout on CPU 1
    mytimeid = 0;

    /* Rest of work */

    UNLOCK(mylock);
}
```

Figure 7: Example of *untimeout* deadlock

requirements must be present. However, in drivers which use watchdog timers, these locking requirements represent the rule rather than the exception.

To describe the locking conditions necessary for this deadlock to occur, first assume that there is a driver spin-lock which is used by the driver to protect its state data, and that this data must be accessed from the driver process-level routines, driver interrupt routine and watchdog timer routine. The problem arises when untimeout is called with the driver lock held and the watchdog timer function has been called and is trying to acquire the driver lock. The semantics of untimeout are defined so that it does not return until either the timeout is safely canceled or has completed running, if it is found running when the call is made. This restriction is necessary because it may be unsafe for the timeout function to continue running after the driver code thinks that it has been canceled. For example, a device close routine could free data structures which the timeout function would attempt to access. Figure 7 shows code on two processors which illustrates the deadlock, where *mytimeid* is the timer id returned from the timeout call which set up the timer.

There are a few solutions to this problem. The one adopted in the multiprocessor DDI/DKI [USL91] is to prohibit the calling of untimeout with the driver lock held. This results in a code structure which does not fit well with some old drivers, as shown in Figure 8. The problem is that the code that needs to be idempotent is not, where idempotency implies that the code can be executed repeatedly without destroying the consistency or correctness of the driver data structures. Restructuring old drivers in this way could involve some extremely messy changes when the driver code is very complicated[1]. The management of *mytimeid* must also be done carefully, since the zeroing of it and the actual untimeout are not atomic. Hence, it cannot be relied on as an indicator that there is no timeout pending.

The Consortium developed a different solution to this problem, which requires only a small change to any timeout routine that might deadlock in this way. The timeout routine must use a non-spinning form of the lock primitive which returns a failure indication when the lock is busy. If the lock is found to be busy, then a new function *untimed* is called, which returns non-zero if an untimeout is waiting for this timeout to finish. If untimed returns non-zero, then the timeout function should return immediately, allowing the untimeout to finish and thus breaking the deadlock. The code with replaces the LOCK call in mytimeout is shown in Figure 9.

Since this problem was cited as an example of a weightless process, are there other solutions, particularly using kernel threads? The basic issue is that committing to the running of the timeout function and the acquisition of the driver lock should be made atomic to avoid any deadlocks or back-offs. If

[1] SVR4 licensees can look at the kernel file io/asy.c as an example of such a driver.

```
myclose(dev)
{
    int save;

    LOCK(mylock);

again:
    /* Do some work - needs to be idempotent */

    if (save = mytimeid) {
        mytimeid = 0;
        UNLOCK(mylock, plstr);
        /*
         * A device interrupt or timeout
         * could now run on another processor.
         */
        untimeout(save);
        LOCK(mylock);
        goto again; /* May need to recheck state */
    }

    /* Rest of work */

    UNLOCK(mylock, oldpri);
}
```

Figure 8: DDI/DKI *untimeout* Fix

the timeout is run as a thread similar to interrupts in Solaris [Eykholt92], then it would have a context to suspend while waiting for the driver lock. In the suspended state, if the thread is killable by untimeout, then the deadlock can be avoided.

```
mytimeout(arg)
{
    while (TRYLOCK(mylock) == FAIL)
        if (untimed())
            return;
    /* mylock is locked */
    /* Do the work */
    UNLOCK(mylock);
}
```

Figure 9: Fix for *untimeout* Using *untimed*

Related Work

There are two significant STREAMS multithreading efforts which are relevant to the SVR4MP effort. The first was done by Sequent [Garg90] and used the most general model, with maximal concurrency. Sequent claims to have achieved robustness with reasonable performance, without giving details on how the problems of too much locking and circular queue dependencies were dealt with. It is possible that they were more willing to restrict the model used by STREAMS modules and modify existing modules heavily to fit their model. They have also identified one of the annoying aspects of the resource callback and timeout facilities used by STREAMS, namely that the *bufcall* mechanism to queue a buffer allocation callback had no *unbufcall* function to cancel a previously issued bufcall when a queue is decommissioned. They defined such a function, which has been incorporated into the multiprocessor DDI/DKI. Unfortunately, most existing modules do not use the unbufcall utility. A similar problem exists for timeouts, where most modules did not cancel pending timeouts when closed. The original Consortium STREAMS locking was actually very similar in design to that described by Sequent, so that the differences between the current Consortium locking and Sequent locking would be pretty much the same as the differences between Version I and Version III of the locking.

The other important STREAMS effort was described informally by Sun [Barton92] and makes good use of the kernel threads in Solaris, as previously described. Their locking is also fine-grained at the queue level, but there is apparently a sub-queue off each queue to hold requests for servicing when a queue is found locked. This approach probably solves most if not all of the likely deadlock scenarios and is similar to attaching a streams coupler module onto each queue.

Conclusions

Multithreading STREAMS proved to be a very difficult and subtle task, which required considerable effort. Several versions of STREAMS locking were implemented, with later versions actually providing less potential concurrency, but having far less overhead and better stability and scalability. This phenomenon has occurred repeatedly within the context of the SVR4MP effort – the tendency was to make locking too fine-grained on the first attempt [Intel92, Peacock92]. The current implementation has no known deadlock scenarios or races in the closing paths. It is difficult to judge whether this means there are actually none left, because most existing STREAMS modules do not, in fact, conform completely to the DDI/DKI specification, especially some of the multiplexor modules. The most likely errors remaining to be discovered are of the "hanging queue pointer" variety, where queue pointers are saved and reused at a later time without regard to whether the queue has been freed.

There were a number of other problems which were solved aside from those described in the paper. In particular, flow control mechanisms required quite a bit of modification to work reliably. Some STREAMS modules required a mechanism whereby they could be forced to run on a specified processor. This problem was solved by surrounding such a module by a pair of *migrator* modules. These modules were very similar to the coupler module described earlier and allowed the scheduling of a queue's put and service procedures on the correct processor.

There are some interesting possibilities for future work in the STREAMS area. The TCP/IP protocols were actually quite difficult to multithread correctly due to the structure of the original implementation. It is probable that some effort to restructure the interactions that take place across the TCP and IP multiplexors could improve the efficiency and scalability of the networking. The IP multiplexor, in particular, may benefit from some internal multithreading to allow parallel processing of packets, since strict ordering at the IP level is not required.

Another area which requires some investigation and improvement is the scheduling of STREAMS procedures. The current queueing arrangement is somewhat clumsy and the STREAMS dispatching code spends quite a bit of time, mostly from the idle process, trying and failing to dispatch service routines because the stream locks are found to be busy. In addition, there are no priorities associated with any STREAMS processing, so the fairness and throughput properties of the scheduling are far from optimal.

Acknowledgments

We would like to thank Intel Corporation for sponsoring the Intel Multiprocessor Consortium and allowing us to spend the last 3 years working on such interesting and challenging problems. We should also acknowledge and thank Cliff Neighbors, who was the first to try and wrap his wits around this large and complex area. Lastly, the reviewers comments helped us produce a paper which is better than it would otherwise have been.

References

[Barton92] Steve Barton. Overhead slides from Solaris 2.0 Technical Birds-of-a-Feather session at USENIX. San Antonio, June 1992.

[Eykholt92] J. R. Eykholt, S. R. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Wekks, D. Williams. Beyond Multiprocessing: Multithreading the SunOS Kernel. *Proceedings of the Summer 1992 USENIX Conference.*

[Garg90] Arun Garg. Parallel STREAMS: a Multi-Processor Implementation. *Proceedings of the Winter 1990 USENIX Conference.*

[Intel92] J. K. Peacock, S. Saxena, D. Thomas, F. Yang and W. Yu. Experiences from Multithreading System V Release 4. *Proceedings of the Third Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS III).* USENIX; March 1992.

[NCR91] M. Campbell, Richard Barton, J. Browning, D. Cervenka, B. Curry, T. Davis, T. Edmonds, R. Holt, J. Slice, T. Smith and R. Wescott. The Parallelization of UNIX System V Release 4.0. *Proceedings of the Winter 1991 USENIX Conference.*

[Peacock92] J. K. Peacock. Files System Multithreading in System V Release 4 MP. *Proceedings of the Summer 1992 USENIX Conference.*

[Ritchie84] D. M. Ritchie. A Stream Input-Output System. *AT&T Bell Laboratories Technical Journal.* October, 1984.

[USL91] Unix System Laboratories and Intel Corporation. *UNIX System V Release 4 Multi-Processor Version 1 for Intel Processors Device Driver Interface/Driver-Kernel Interface (DDI/DKI) Reference Manual.* Draft version.

Author Information

Sunil Saxena has worked at Intel as a Consultant for the last 3 years as a member of the Advanced System Architecture team and Intel Multiprocessor Consortium. He has worked part-time at Unisys to provide debugging support for the SVR4-MP Unix. Prior to that, he worked at Acer/Counterpoint on the multiprocessor implementation of System V for the Motorola 680x0 based platforms. He has worked on numerous Unix based platforms in the areas of porting Unix, debugging tools, performance enhancements and fixing bugs. He graduated from the University of Waterloo in Ontario, Canada with a Ph.D. in Electrical Engineering in 1980, and a Master of Mathematics in Computer Science in 1976. In 1975, he graduated from the Indian Institute of Technology, Delhi, India, with a B.Tech. in Electrical Engineering. He can be reached via U.S. Mail at Uni Plus Inc, 156 Carlow Ct, Sunnyvale, CA 94087 and electronically at sunil@chili.intel.com.

Kent Peacock has worked at Intel as a Consultant for almost 3 years as a member of the Intel Multiprocessor Consortium. Prior to that, he worked at Acer/Counterpoint developing a multiprocessor implementation of System V and the Acer/Counterpoint Fast File System, which is now part of SCO UNIX. He has worked on design and implementation of 5 multiprocessor systems since 1978, and has dabbled in performance tuning, C compilers, multiprocessor debugging tools and graphics applications. He graduated from the University of Waterloo in Ontario, Canada with a Ph.D. in Computer Science in 1979, having previously completed a Master of Mathematics in Computer Science in 1975. In 1974, he graduated from the University of Manitoba, in Winnipeg, Canada, with a Bachelor of Science in Electrical Engineering. He can be reached via U.S. Mail at 1747 Fanwood Ct., San Jose, CA 95133 and electronically at kentp@stps18.intel.com.

Fred Yang is a software engineer at Intel. He has worked as member of Intel Multiprocessor Consortium for the last 3 years. He worked on process management, TCP/IP, NFS and STREAMS of SVR4-MP Unix project. Prior to that, he worked on various MACH projects for Intel multiprocessor platforms. He received his MSCS from the University of Texas at Arlington in 1983, and BS in Applied Mathematics from Chau-Tung National University, Taiwan, in 1977. He can be reached electronically at fred@fred.intel.com.

Vijaya Verma has worked at Intel as a member of the Intel Multiprocessor Consortium from Wipro in the area of Networking and Streams. She has been working at Wipro for the last 6 years in the areas of Communications and Protocol development under Unix. She graduated from Indian Institute of Technology, Bombay, India, with an M. Tech. degree in Communication in 1983. In 1981, she graduated from MS University, Baroda, India, with a B.E in Electronics and Communication. She can be reached at Wipro Infotech Ltd., 88 M G Road, 56001 Karnataka, India, and electronically at wipro!vv@uunet.uu.net.

Mohan Krishnan has worked at Intel as a member of the Intel Multiprocessor Consortium from Wipro in the area of Networking and Streams. He

has been working at Wipro for the last 5 years in the areas of Device Drivers and IPC under Unix. He graduated from Indian Institute of Technology, Madras, India, with a M. Tech. degree in Computer Science in 1988. In 1986, he graduated from Calicut University, India, with B.Tech in Electronics and Communication. He can be reached at Wipro Infotech Ltd., 88 M G Road, 56001 Karnataka, India. His electronic mail address is wipro!mohn@uunet.uu.net.

WARLOCK – A Static Data Race Analysis Tool

Nicholas Sterling – SunSoft, Inc.

ABSTRACT

Concurrent programming is becoming available to the masses, bringing with it the potential for new types of errors such as deadlocks and data races. This paper describes a static data race analysis tool less ambitious than most, written for use with SunSoft's Solaris operating system. The basic algorithm is described, and a sample use of the tool is discussed. Some complicating factors of real code are presented, along with the means chosen to deal with them. The current status of the tool and some preliminary experiences are discussed.

Introduction

While concurrent programming has been explored in research communities for decades, it has only been available from vendors targeting various specialized markets. We are approaching a milestone in that such capabilities will soon be widely available on mainstream workstations. SunSoft's Solaris operating system is now multithreaded and allows application software access to the multithreading model through library calls. A new POSIX standard defines primitives which allow the development of portable multithreaded applications [POSIX]. Major software vendors are beginning to make use of threads in order to improve performance on both uniprocessors and multiprocessors.

In the multithreading model, a program consists of one or more threads of control which share a common address space and most other program resources. The various threads of control may execute in parallel on a multiprocessor; even on a uniprocessor the interleaving of progress on various threads is non-deterministic. Threads must acquire and release locks associated with shared data in order to reliably produce the intended results; where they fail to do so, data may become corrupted. This is a type of "data race" – a situation in which a program may produce different results when run repeatedly with the same input.

The benefits of multithreading technology are obvious: increased parallelism on multiprocessors, and the ability to better express the asynchronicity inherent in many problems. Multithreading can even improve the performance of programs run on uniprocessors. But the technology also brings with it significant challenges. Research on concurrent programming clearly shows that debugging and testing of concurrent programs is more difficult than it is for sequential programs, largely because of the non-determinism caused by differences in execution order of code within the various threads of control [MH89].

Data races are easy problems to introduce – simply accessing a variable without first acquiring the appropriate lock can cause one – and they are generally very difficult to find. Visible symptoms of a data race generally manifest only if two threads access the improperly protected data at nearly the same time; hence a data race may easily run for months without showing any signs of a problem. It would be extremely difficult to exhaustively test the various concurrency states for even a simple multithreaded program, so conventional testing and debugging are not an adequate defense against data races.

Furthermore, because threads which fail to use locks properly can generally interfere with one another at various points in their execution, the symptoms may be different each time a problem does occur. To make things worse, post-mortem analysis of a crashed program will probably show which data are in an invalid state, but will generally offer no clues as to how they reached that state. When such problems are discovered in the field, which is likely for highly intermittent problems, customers often do not have time to help debug the problem.

Considerable research has focussed on new tools to address these problems [MH89]. One technique is to analyze the program source code, looking for potential problems. This technique is called "static analysis" to distinguish it from dynamic techniques, which involve running the program and analyzing its actual run-time behavior. Since dynamic analysis requires that the program be forced through the program states of interest, its use with multithreaded software is problematic.

Depending upon the semantics of the source language, static analysis can point out many types of potential problems, such as references to uninitialized variables, waiting for threads of control which have already terminated and been released, and references to variables with non-deterministic values [TO80]. Such analysis is often based on the creation

of a concurrency graph, in which each node represents a unique possible state of the entire program – i.e. a unique set of locations for the various threads of control in their own individual flow graphs [Tay83]. For large programs with substantial concurrency this technique suffers from a combinatorial explosion in the size of the graph, although mechanisms can often be applied to deal with this explosion [YT88].

The tool described in this paper, called "warlock", is a static analysis tool less ambitious than the research activity mentioned. It is interesting primarily for its relevance to a widely available platform for concurrent programming and for the tradeoffs made in order to produce a meaningful tool in the context of the language and multithreading model. This paper is not so much about the science of static analysis as it is about the engineering of this particular tool.

Locking in Solaris

It is assumed that the reader has a rudimentary familiarity with the model of multithreading implemented in Solaris [Pow91, Kle92]. A brief review of the locking facilities follows.

Solaris supports simple mutex locks which are acquired and released with the calls:

```
mutex_lock(mutex_t*)
mutex_trylock(mutex_t*)
mutex_unlock(mutex_t*)
```

These are the most efficient locks to use for simple mutual exclusion. Mutex_trylock() returns a failure code rather than blocking the thread if the lock is unavailable.

Also supported are readers/writer locks, which decrease contention by allowing multiple readers to hold the lock at one time. The following calls are supported:

```
rw_rdlock(rwlock_t*)
rw_wrlock(rwlock_t*)
rw_tryrdlock(rwlock_t*)
rw_trywrlock(rwlock_t*)
rw_unlock(rwlock_t*)
```

Rw_tryrdlock() and rw_trywrlock() return a failure code rather than blocking the thread if the lock is unavailable.

Counting semaphores and condition variables are also available but, for reasons to be discussed later, warlock ignores them.

The granularity of data protected by locks is entirely up to the developer, and in practice varies considerably depending upon the amount of contention expected, the weight of the operations to be performed, and other factors. In one case an entire library may be completely serialized through the use of a single lock, while in another case a single structure may contain multiple locks, each of which protects a few members of the structure. Very commonly a structure will contain a single lock to protect its members, or a single lock will protect a collection of some kind, such as a linked list.

Warlock Background

Warlock was initially created as an aid to developers of the kernel and programs which comprise the Solaris operating system. A tool similar to the well known UNIX lint utility was visualized, with makefile targets set up to run the tool automatically. Also a static analysis tool, lint serves as a familiar model for C programmers.

For such a tool to do its analysis using a concurrency graph would have been problematic. Multithreaded kernel code is almost completely event-driven; so is a high percentage of application software, notably servers and window-based tools. In event-driven software, code is typically structured so that it is almost all concurrently executable. Such a breadth of concurrency, and the sheer size of some of this software, would tend to make the concurrency graphs intractably large.

Furthermore, a scalable mechanism was needed. Much of the code to be analyzed consists of libraries, which one would like to analyze independently of any particular program's use of them. Moreover, the development process itself involves a large number of engineers who cannot be called together to participate in the analysis procedure. For these reasons, as well as the time it would take to perform a large analysis, the tool would have to allow the code to be analyzed in smaller pieces.

Also, it was felt that the most interesting information to be gained from such a tool was potential data races. They were felt to be the problems most likely to remain undetected in code which seemed to work. Deadlocks were also of interest, but a separate internal tool had already been written to help find them.

As a result, a simpler approach to finding data races was chosen. Rather than explicitly looking for data races using a concurrency graph, which requires so much processing, warlock simply checks for consistency in the code's use of locks. In so doing it detects what was felt to be the most common cause of data races: failure to hold the appropriate lock while accessing a variable.

Addressing the problem in this manner presupposes that each lock is associated with, or "protects," an unchanging set of variables. While there is no requirement that locks be used in this manner (one could protect a variable with one lock during one phase of a program's execution, and then with a different lock during another phase), they almost always are. With a few degenerate exceptions, properly protecting variables when the correct method

changes over time would be highly error-prone, especially in code which is nearly all concurrently executable. So warlock generally assumes that locks are used consistently, and provides special mechanisms for dealing with exceptions.

Warlock discovers locking problems basically by tracing the execution of every path through the code, noting which locks are held each time a variable is accessed. A symbol table is maintained for variables, and for each entry in that table a list is maintained of locks consistently held when that variable was accessed. After all execution paths have been traced, if a variable's list of locks consistently held is empty, then – as far as warlock can tell – that variable is not properly protected by a lock, and any code which accesses that variable is subject to data races.

Consider the following simple (and contrived) example:

```
mutex_t lock1, lock2;
int var1, var2;

func1()
{
    mutex_lock(&lock1);
    if (var1++)
        func3();
    mutex_unlock(&lock1);
}

func2()
{
    mutex_lock(&lock2);
    var2++;
    mutex_unlock(&lock2);
}

func3()
{
    var2 = 9;
}
```

Warlock recognizes that func1() and func2() are not called from anywhere, so it traces the execution of each, descending into func3() when it is called to trace its execution as well. Warlock traces parts of func1() twice, once with the expression controlling the if statement true and once with it false. When all paths have been traced, warlock discovers that lock1 was held every time var1 was accessed, but there is no lock which was held every time var2 was accessed (during one access lock1 was held, and during the other lock2 was held). Therefore uses of var2 are potential data races.

With real software, there is much to complicate this basic algorithm. There are also other useful results which can be produced.

Clearly, warlock does not detect all data races. It is possible (but probably relatively rare) to write code which produces non-deterministic results even while all accesses are properly protected by locks, as does the following code:

```
mutex_lock(&foo->lock);
count = foo->count;
mutex_unlock(&foo->lock);
...
mutex_lock(&foo->lock);
foo->count = count + 1;
mutex_unlock(&foo->lock);
```

Furthermore some variables, while protected consistently, are protected by a mechanism more complex than simply acquiring a lock. For example, some data is effectively protected by semaphores and condition variables, while warlock only analyzes the use of mutex locks and readers/ writer locks. However, mutex and readers/writer locks are typically used to protect most data because of their greater efficiency.

Mutex and readers/writer locks are suitable for analysis since a thread uses them by bracketing code with calls to acquire and release the lock. Semaphores, on the other hand, may be used in a variety of ways, frequently with one thread doing a P operation and another thread the corresponding V operation:

```
produce()
{
    static int next_empty_buffer = 0;
    sema_p(&empty_buffers);
    fill(next_empty_buffer);
    if (++next_empty_buffer >=
                    NUM_BUFFERS)
        next_empty_buffer = 0;
    sema_v(&full_buffers);
}

consume()
{
    static int next_full_buffer = 0;
    sema_p(&full_buffers);
    if (++next_full_buffer >=
                    NUM_BUFFERS)
        next_full_buffer = 0;
    empty(next_full_buffer);
    sema_v(&empty_buffers);
}
```

This makes it more difficult to identify which lines of code are actually under the protection of a semaphore; the rules could be different for each abstraction created using semaphores. Furthermore, it is difficult to distinguish between different elements of an array in static analysis. For example, it would be difficult for a static analysis tool to discern that in routine consume() above, because the call to empty() follows the increment of next_full_buffer, the wrong buffer is being accessed. As a result, the program might attempt to empty and fill a given buffer at the

same time. Counting semaphores are often used with arrays in this manner.

Similarly, condition variables are often not used in a simple bracketing manner. In those cases where semaphores and condition variables are used in a bracketing manner, it is sometimes possible to modify the code so that warlock sees the locks as mutex locks, allowing the code to be analyzed.

Using Warlock – An Overview

In order to trace the execution of the program, warlock needs a representation of each function showing its activity relevant to the analysis. A modified version of SunPro's ANSI C compiler produces for each .c file not only a .o file, but also a .wl file. This .wl file contains information about the flow of control in each function, as well as each access to a variable or operation on a mutex or readers/ writer lock.

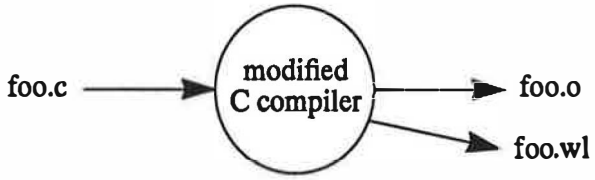


Figure 1: Creating Warlock Information Files

When all of the code to be analyzed has been compiled using this modified compiler, the user invokes warlock to perform the analysis. Used in this way, warlock is an interactive program. The user issues commands to load the relevant .wl files and gives other hints which will improve the analysis. In practice, these commands are generally stored in a start-up file and are executed automatically when warlock is invoked.

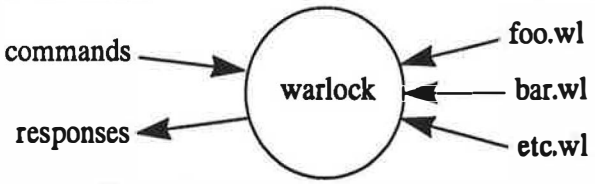


Figure 2: Interacting with Warlock

Then the user directs warlock to perform the analysis. The analysis may produce messages saying that a lock was released when it was not held, that a covered lock was acquired while its cover was not held, or the like. If a program has been running successfully, these particular messages usually don't reflect true problems in the code, but rather limitations in warlock's ability to understand the code. Through the use of conditional compilation one can present to warlock a somewhat simplified version of troublesome code fragments, allowing warlock to conduct a meaningful analysis.

Once the analysis completes without such errors, the user asks to see which variables are not consistently protected by locks. The user may make assertions to warlock about which variables are supposed to be protected by a lock and about which locks are supposed to be held whenever a function is called. Running the analysis with such assertions in place will show the user where the assertions are violated. Again, these violations may represent limitations in warlock's ability to understand the code; or they may represent actual problems in the code.

While the separate, interactive back end provides the most powerful access to warlock's capabilities, warlock can be used in a more lint-like manner as well. The code to be analyzed can be annotated with special warlock directives which provide information warlock cannot glean from the code itself. Then a script can be run which compiles the source using the modified compiler, performs the analysis, and reports potential problems. Ultimately SunSoft plans to instrument all kernel code with appropriate directives and perform periodic analyses of the entire kernel by running the lint-like script from makefiles.

While the interactive use of the back end is interesting for its ability to answer questions about the behavior of a program relevant to locks, this paper will not discuss that interface. Instead the focus is on the use of source annotation to influence the analysis. By annotating common header files, the burden of the individual developer in running warlock is decreased and useful documentation is provided at the same time.

Some Complications

The earlier example was straightforward, but complications abound in real code. In fact, most of the effort both in developing warlock and in using warlock results from complications rather than the basic algorithm. This section discusses some of these complications and how they were addressed.

Frequently, the mechanism chosen for dealing with a complication was to make available to the user of warlock a way of annotating the program source code to inform warlock of things which it cannot glean from the source itself. These annotations, often called "directives," are in the form of comments with special keywords which the compiler has been modified to recognize. Source-annotation serves the same purpose in lint.

The intent was to have the comments provide documentation which would help a developer looking at the code for the first time. In effect, standard ways of expressing such ideas about locks have been defined, and the compiler modified for warlock has been taught to recognize them.

The autocacher is invoked in much the same way as the automounter. A sample configuration table is shown in Figure 1.

The lines beginning with a # are comments. The first directory name is the directory which will be emulated. We require that the full path name be given. The next entry is the remote directory, and the name following the ":" is the local cache. None of the components of the local tree need exist: directories are created as needed. If components of a local tree exist but are of the wrong type, which can happen when a file in the remote tree turns into a directory, the local tree is modified so that it conforms to the remote tree. This may involve deleting and recreating files and directories.

The autocacher will emulate a traditional directory structure that is a mirror of the remote file system. In each directory there can be other directories, soft links (to local or remote files), or regular files. As described above, the files are actually three-valued, depending on whether the autocacher has a valid cached copy; is emulating a file; or is returning a pointer to a remote copy. Files that are accessed in the emulated directories will return soft links to files in the local cache if at all possible. The files in the local cache must meet several criteria described in greater detail below. If there is no way to satisfy the request with a soft link to the local copy of the file, then the autocacher will return a soft link to the remote copy. Note the advantage this has over caching file systems such as Andrew: the fallback mechanism will function, albeit more slowly, rather than fail to access the file at all. In fact the fallback mechanism is to simply use the remote NFS server for files.

One problem that can occur is that NFS will attempt to cache attributes of a file. This caching is done in order to reduce network traffic. In the case of emulated files, this can result in more READ requests being directed to the autocacher than are necessary, as NFS will not timeout the cache for a very long time, and thus will not be redirected via a soft link to the local cached copy. To ensure that the workstation does not maintain autocacher entries in its NFS client cache for long periods, the autocacher always modifies the atime, mtime, and ctime values it returns for files to be as recent as possible.

Ramifications of the NFS Protocol for the Autocacher

NFS is a stateless protocol. This means that the autocacher can not determine, from the NFS operations that it handles, whether a file is being opened for reading or simply being examined (via an *ls* command, for example). The sequence of operations preceding an actual read will consist of an NFS LOOKUP request followed by a GETATTR request. The only way to determine that a read is occurring (meaning that a file has been opened for read, and should be cached) is for the autocacher to field the NFS READ call itself. Once the autocacher has begun fielding READ requests, and until the next LOOKUP request occurs, the autocacher must support all the READ calls itself. Once the next LOOKUP occurs, the autocacher can emulate a soft link instead of a file as there is now a local copy of the file that is valid, and it no longer needs to support READ operations for that file.

Once the autocacher has started supporting READ operations, it may keep the local file open for efficiency reasons, it being inefficient to open, read the file, and close it again for each READ request. A significant problem is that the next LOOKUP may not occur for a very long period of time – in fact it will not occur until the next time some activity occurs on the file, which in theory could be forever. Once the autocacher is asked to support a READ, however, it must open the local file and keep it open for subsequent read requests. One can construct scenarios where the autocacher runs out of file descriptors because too many files are open.

In practice this problem may be dealt with in several simple ways. First, short files (currently this means files smaller than 32K bytes) are read into memory and the file is closed. Thus, most files are actually not open for the period during which the autocacher is supporting reads on that file. Second, there are high water marks in the code such that when the number of open file descriptors exceeds the water mark, files that have been open for a long time and which have not been read for a long time are closed. Finally, there is a desperation mode in which file descriptors are scavenged in LRU order, and which could in extreme circumstances result in the opening and closing of a file for each read operation. These are simple heuristics which in the past two and a half years have not failed – we have never exceeded 64 open file descriptors in actual use[2] We have tested the desperation code but not exercised it in actual use.

When the autocacher reads its configuration file, it builds an internal representation of the files it has to cache. The representation has three components: the remote name of the file being cached; the local (i.e., "cached") name of the file; and a stat buffer derived from a stat of the remote file.

When a LOOKUP request for a file is received, the autocacher determines the state of the cache. There are three possible cases:

- The local file is in the cache and up-to-date. In this case, the LOOKUP returns a soft link to the local cached file.
- The local file is in the cache, but out-of-date. The autocacher determines that a local file is

[2]The current reigning champion is Interleaf, which opens at least 56 files when a document is opened for editing

Uninteresting variables

Some variables are of no interest. Const variables and thread-local variables (identified to the compiler using a pragma) are ignored automatically by the modified compiler; no information about them is stored in the .wl file.

Automatic variables are ignored as well. An automatic may only be accessed from other threads via a pointer. Pointers are not ignored, so warlock handles this situation correctly unless the defining thread accesses the variable without using the pointer. A future release will provide support for this case.

The user may want other variables ignored as well, for various reasons. A variable might effectively be protected by a semaphore, while warlock doesn't analyze the use of semaphores. Some variables are used in such a way that no locking is required. For example, it is common for a variable to be written once before any other use, and only read thereafter.

Embedding the comment

```
/* VARIABLES PROTECTED BY
                "sem1": v1, v2 */
```

in the program source informs warlock that variables v1 and v2 are protected by some means other than a mutex or readers/writer lock (in this case semaphore sem1), and should therefore be ignored.

Initialization code

Earlier it was stated that in practice variables are protected consistently by locks. One degenerate exception to this is initialization code. A program starts life as a single thread, and often initializes variables before creating other threads. During this time, data races are impossible since only one thread is running, so the program may safely access data without holding the associated locks. Similarly, programs often finish with one thread waiting for all of the others to exit, and then doing some final work such as printing results; again, there is no need to acquire locks during this period. Special comments express this:

```
main()
{
  /* NO OTHER THREADS ARE RUNNING */
  initialize();
  /* OTHER THREADS ARE RUNNING */
  do_work_with_multiple_threads();
  /* NO OTHER THREADS ARE RUNNING */
  print_results();
}
```

Warlock ignores accesses to variables during periods when no other threads are running.

There is a second aspect to the initialization exception. A structure allocated off the heap is not globally accessible until the thread which allocates it places a pointer to it someplace where it can be seen by another thread. Commonly a routine will, after allocating a structure, initialize it without holding the corresponding locks. Again, special comments apprise warlock of the situation:

```
extern struct foo *global_foo;
struct foo *p;
p = (struct foo *) malloc(sizeof(*p));
/* LOCK UNNEEDED BECAUSE
      "p not shared": p->lock */
p->count = 0;
```

Pointers to functions

When a function is called through a pointer, warlock needs to descend into each of the functions which might actually be targeted by the call. Where the function pointer is part of a structure, the modified compiler watches for initializations of such structures and automatically records the initialization values as possible targets.

For all other function pointers, the user must enumerate the possible targets using a special comment:

```
void (*f)();
/* FUNCTIONS CALLED THROUGH POINTER f: a, b */
```

Even with this facility, function pointer arguments represent a problem. Consider the C library function qsort(), which takes a pointer to a comparison function. Certainly the implementation of qsort() could not contain the directive, since the callers of qsort() are different for each body of code analyzed. But for the calling code to use the directive, it would have to know the name of the variable qsort() uses to receive the function pointer. Furthermore, the caller would have to know whether qsort() passes the pointer to other functions, and the names those functions use to receive it.

However, this situation is simplified by writing for each library a set of functions which represent the behavior of interest to warlock in a consistent, simple manner (similar to the use of lint libraries). For example, the warlock version of qsort() might look like this:

```
qsort(char *a, int b, int c,
                    int (*qsort_FP)())
{
    (*qsort_FP)();
}
```

Assume that by convention such routines are written to accept function pointers in variables called "<function>_FP", and that such routines do not in turn pass those function pointers on to other functions. Then developers can enumerate targets of function pointer arguments as follows:

```
/* FUNCTIONS CALLED THROUGH POINTER
    qsort_FP: foo, bar */
```

This problem may someday be addressed automatically by implementing in warlock a limited form of data analysis to keep track of function pointers passed as arguments.

Incidentally, the majority of functions in the C library have no behavior of interest to warlock, and hence do not require that such counterparts be written. A warlock version is needed for a function only if it takes a function pointer as an argument, has locking side effects, or assumes locks are held on entry.

Anonymous data

Warlock, like most static analyzers, cannot distinguish between different instances of a data type accessed through pointers and/or array indexing. For structures, warlock employs the usual solution of treating all such instances of the structure as a single instance [Tay83]. In its reporting warlock refers to member "mbr" of an anonymous instance of the structure with tag "tag" as "tag::mbr". This notation is borrowed from the C++ language.

Unfortunately, the C language allows a structure to be defined without assigning it a tag. In this event the compiler generally makes up a tag name for the structure. Warlock could not use the name fabricated by the particular compiler upon which warlock was based, since the name assigned was not consistent across programs sharing the definition of that structure. Therefore warlock assigns its own name for the tag, creating it from the filename and line number in which the structure was defined. For example, the tag name created for a tagless structure defined on line 10 of file "x.c" would be "x.c@10". Clearly this approach works for structures defined in include files but fails if the definition is simply copied into multiple files.

Anonymous simple types are ignored by warlock. In the following example, the warlock compiler makes a record of the access to foo::a, but not for the access to *ip:

```
struct foo {
        int a;
};

bar(struct foo* foop, int* ip)
{
        foop->a = 2;
        *ip = 3;
}
```

The write to *ip is every bit as subject to data races as the write to foop->a, so it would be helpful to have warlock check whether accesses to it are consistently protected. However, while all foo::a are protected by a single lock (or at least, warlock assumes so), it is unlikely that all anonymous ints are protected by a single lock. Therefore if warlock were to include such accesses in its analysis, it would almost certainly report them as errors.

Scope of identifiers

C, like many block-structured languages, allows a variable to become hidden by the introduction of another variable by the same name in a more immediate scope. Within the program, the scope rules determine which variable by that name will actually be accessed, and the others are simply inaccessible. By embedding directives in source in the form of comments and then parsing the directives along with actual program code, scope rules can be applied to names within directives.

Function names used in the aforementioned FUNCTIONS CALLED THROUGH POINTER directive are an exception. C allows the use of function names for which no declaration exists – that is, names which are not in scope. These names cannot be checked for validity by the modified warlock compiler. Moreover, the user needs the ability to specify which of several possible functions by that name is intended. Therefore the syntax '"filename":function' is allowed for a function name.

Union members

When a member of a union is accessed, all other members of that union are accessed as well. For this reason union member names are all recorded using a single name: %. This causes accesses to various members of a union to be treated as accesses to the same variable. The implementation errs on the side of caution when some of the union members are structures, but this does not seem to cause trouble.

Loops and recursion

If warlock finds itself looping or recursing while tracing the execution of a program, it ends the trace of that path through the program so as not to loop or recurse forever. This can cause warlock to miss possible problems. For example, in the code

```
for (i=0; i<10; i++) {
    mutex_lock(&foo_lock);
    foo = foo + i;
}
```

warlock will only trace one iteration of the loop, and therefore will not flag an error that an attempt is made to acquire foo_lock when it is already held.

Data dependencies

Currently warlock makes no attempt whatsoever to keep track of the values of variables. This can cause warlock to analyze paths through the code which could not really be taken. In the following code fragment, for example, if the lock is acquired, then it is released, and if it is not acquired, then it is not released.

```
Bool we_locked_it = FALSE;
if (...) {
    mutex_lock(&lock);
    we_locked_it = TRUE;
}
```

```
  ...
if (we_locked_it)
    mutex_unlock(&lock);
```

But because warlock does not keep track of the value of variable we_locked_it, warlock sees four possible paths through this code rather than two. In one of those paths the program unlocks the lock without having locked it – clearly an error – and in the other the program leaves the lock locked, causing the function to appear to have inconsistent side-effects on locks.

Such data dependencies can be circumvented by presenting to warlock a simpler view of the code. This can be done using macro WARLOCK, which is defined during the creation of the .wl file but not during the creation of the object file. The example above might be altered to look like this:

```
Bool we_locked_it = FALSE;
#ifdef WARLOCK
mutex_lock(&lock);
#else
if (...) {
    mutex_lock(&lock);
    we_locked_it = TRUE;
}
#endif
  ...
#ifdef WARLOCK
mutex_unlock(&lock);
#else
if (we_locked_it)
    mutex_unlock(&lock);
#endif
```

Dealing with data dependencies is one of the more problematic aspects of using warlock. Not only does it require that the user modify source code, but the resulting code is also harder to read. In the future, limited forms of data analysis may be employed in order to automatically handle some of the common problems.

Partial analyses

Warlock traverses the call graph for the code being analyzed to identify the "root" functions – those which are not called from any other function being analyzed. Main() and signal handlers would typically be root functions for a user program. The trace of the program's execution proceeds from each of the root functions, down into the functions it calls.

Because one frequently wants to analyze a library or a subsystem of a very large program (like the kernel), a non-root function may also be called from outside the set of functions currently being analyzed. Such a function should be considered a root function in the context of the set of functions being analyzed. This situation arises mostly with analysis of libraries, where a function which is part of the library's public interface is also called by a function within the library.

One could envision a directive which informs warlock that a function should be treated as though it were a root, but that would not solve the problem. The need to identify a particular function as a root function comes not from the nature of the function itself, but rather from the functions one has chosen not to include in the analysis. An annotation would cause the function to be treated as a root in any analysis, whereas the function really needs to be treated as root or not depending upon which other functions are analyzed along with it.

One way of dealing with the problem is to write a dummy function which calls each of the public library functions and analyze it along with the library:

```
main() {
    public_func1();
    public_func2();
    ...
}
```

Another problem with partial analyses arises when a function assumes that a lock is acquired by the caller. If no caller is provided in the analysis, it will appear to warlock that the function accesses variables without holding the appropriate locks. A special comment exists to communicate the function's assumption to warlock:

```
foo(struct bar* b)
{
  /* LOCK HELD ON ENTRY: b->lock */
```

If the function is a root, warlock automatically acquires the specified lock before it begins the analysis of this function as a root. If the function is ever called from another function, warlock checks that the specified lock is held at the time the call is made.

Hierarchical locking conventions

A readers/writer lock may be used to control access to other locks, allowing one to set up lock hierarchies. In warlock parlance, the controlling lock is said to "cover" the other locks. While holding the cover for write access, it is unnecessary to acquire any of the covered locks. It is an error to hold a covered lock while not also holding its cover for at least read access. This technique is sometimes used, for efficiency reasons, to control access to a set of related data structures. For example, in the following code, a readers/writer lock controls access to a linked list of foo_t structures, each of which may be individually locked as well.

```
delete_foo(foo_t *foo) {
  /* lock the entire list */
  rw_wrlock(&list_lock);
  <prepare foo for deletion>
```

```
    /* unlink; no need to
                    lock prev/next */
      <unlink foo>
      rw_unlock(&list_lock);
}
update_foo(foo_t *foo)
{
      /* have to get read
                  access to list */
      rw_rdlock(&list_lock);
      /* now lock that foo and
                      update it */
      mutex_lock(&foo->lock);
      <update foo>
      mutex_unlock(&foo->lock);
      rw_unlock(&list_lock);
}
```

Deleting a foo_t requires that other threads be prevented from using the list; acquiring list_lock for write access accomplishes this, obviating locking individual foo_t structures in the list. Updating a foo_t, on the other hand, does not preclude other threads from using the list – but it does require that other threads be prevented from using the foo_t being updated. Hence it is necessary to acquire the mutex lock protecting that entry. But before the foo_t itself can be locked, we must hold list_lock for read access.

Such hierarchical use of readers/writer locks is simply a convention – there is nothing in the code to indicate to warlock that this relationship exists, so warlock reports that the variables protected by such locks are not consistently protected. A special comment informs warlock of the relationship:

```
/* LOCKS COVERED BY list_lock:
                      foo_t::lock */
```

Given this information, warlock knows to expect different locking behavior with these locks. Moreover, warlock can verify that no covered lock is ever held while the cover is not held.

Functions with locking side effects

Part of warlock's analysis involves calculating the net side effects each function has on locks. It is easy to accidentally code a function to have a locking side effect:

```
foo()
{
      mutex_lock(&lock1);
      if (phase_of_moon == BLUE)
          return;
      do_something();
      mutex_unlock(&lock1);
}
```

In the above example, the programmer has probably forgotten to unlock lock1 in the conditional exit. Functions which intentionally leave locks in different states from their entry states do occur, but they are rare, so it is important for warlock to be able to check whether such side effects are valid. Special comments inform warlock of the programmer's intent:

```
/* LOCK ACQUIRED AS SIDE EFFECT: p->lock */
/* LOCK RELEASED AS SIDE EFFECT: lock1 */
```

Any discrepancy between the side effects declared in this manner for a function and the side effects warlock computes for the function is reported as an error.

Conditional locking calls

Some of the locking calls recognized by warlock are not guaranteed to succeed; their return value indicates whether or not the operation was done. If that return value is used to control branching, warlock has to rewrite the code in terms of unconditional operations. For example,

```
if (mutex_trylock(&lock1) == 0) {
      /* success */
} else {
      /* failure */
}
```

gets rewritten as

```
if () {
      mutex_lock(&lock1);
      /* success */
} else
      /* failure */
}
```

Because warlock does no data analysis, there are limits to warlock's ability to decipher the use of such return codes. For example, warlock warns that it can't figure out the following:

```
int return_val = mutex_trylock(&lock1);
if (return_val == 0) {
      /* success */
} else {
      /* failure */
}
```

Unprotected sampling

Consider a garbage-collector which examines reference counts and frees objects to which no references remain. Although updating the reference count requires that a lock be held, the garbage-collector need not hold the lock in order to sample the reference count.

Warlock does not provide direct support for handling such sampling. In order to stop warlock from complaining about the unprotected read, one must employ conditional compilation to change the code warlock sees.

It would not be sufficient to simply provide a way of telling warlock that a particular variable may be safely read without holding a lock. When such a

variable is updated, a lock must be acquired before reading the variable and held until after the variable is written. Warlock would need to be able to tell when a read and write of a variable are part of an update – that is, when the value written is a function of the value read. This is currently beyond warlock's capabilities.

Status and experience

The ability to influence warlock's analysis through source annotation was only recently added to warlock. Most use of warlock involved the interactive back end, which is somewhat less friendly. Considerable development has been done since warlock's first uses, spurred by the difficulties found.

Even so, it is appropriate to think of the existing tool as a prototype for something larger. Too many code changes are required to get warlock to give a program a clean report; various forms of data analysis will be required to reduce the use of comment directives and #ifdef warlock. Warlock currently only analyzes C source, while the use of C++ is growing. Deadlock detection needs to be added to warlock so that separate tools need not be run on all code.

Reaction to warlock has been mixed. Some users have praised it highly, reporting that it led them to problems in their code which would otherwise have been very difficult to find. Others have had trouble understanding what they need to do to their code to eliminate the spurious messages. Also, as one might expect, busy developers have difficulty finding the time to invest in warlock. About a dozen developers have chosen to try warlock during its development, and interest seems to be growing gradually as the tool matures.

Certainly warlock has the potential of catching errors which would otherwise be found in the field, and only after considerable investigation. After an X-windows library was made safe for multithreading use, warlock analysis resulted in eight further changes to the code. On the other hand, part of the kernel's virtual memory subsystem was run through warlock and, after all the complaints had been investigated, no true errors were discovered. While this latter exercise did not improve the code, it raised the developer's confidence in the code's correctness. No doubt the fact that the virtual memory system had been heavily exercised for months made it less likely to contain errors.

Prior to the annotation capability, it typically took a developer four to eight days to read about warlock and then use it successfully with a significant body of code, such as a complex driver or a library. The source annotation capability might improve this situation slightly by providing a more natural interface. Also, once source annotation is in place, standard header files can be annotated once and for all, reducing the workload for individual developers. Similarly, warlock versions of libraries can be written once for all to use.

As one might expect, many of the problems found by warlock are simply cases where a block of code which primarily manipulates one data structure has embedded in it one easily missed reference to another data structure, for which the appropriate lock is not held.

Other problems result from overly aggressive attempts to improve the performance of code. In order to obtain the best performance on a multiprocessor, it makes sense to hold locks for the smallest interval possible. In some cases it is possible to avoid acquiring a lock at all for certain code paths. Occasionally, though, it takes a bit of thought to understand what can safely be done in parallel, and one can easily go a bit too far. In an X-windows library a routine which unrealizes a widget was modified to look something like this simplified version:

```
unrealize_widget(Widget *widget)
{
    if (!widget->realized)
        return;
    mutex_lock(&lock1); /* added */
    <unrealize widget and
                free structure>
    mutex_unlock(&lock1); /* added */
}
```

Clearly the routine was designed to correctly handle multiple calls to unrealize a widget. However, in a concurrent environment two threads could make the call at about the same time, and both threads could pass the test at the entrance to the function, resulting in one of the threads trying to use and free a structure which had already been freed. The fix, of course, was simply to acquire the lock before checking to see whether the widget is currently realized.

The following example depicts another recurrent theme:

```
get_bar()
{
    mutex_lock(&lock1);
    foo = ...
    ...
    mutex_unlock(&lock1);
    return foo->bar;
}
```

Depending upon the circumstances, it may or may not be safe to access foo->bar without holding the lock. In some cases it seems surprisingly difficult to decide.

Recall that warlock detects locking side effects, as this function exhibits on its conditional return:

```
foo()
{
    mutex_lock(&lock1);
    if (phase_of_moon == BLUE)
        return;
    do_something();
    mutex_unlock(&lock1);
}
```

It is somewhat surprising that no errors of this type have been found to date. Perhaps this is simply due to the fact that there is run-time checking to flag a second call to mutex_lock() by a given thread as an error, so such errors would tend to be caught quickly. By contrast, code which suffers from data races can execute correctly for years before a problem manifests, and when it does manifest there may be little to indicate the source of the problem.

Future work

Near-term efforts for the project involve utilizing the annotation capability in system header files and writing warlock versions of common libraries. It is time to provide an environment, using the functionality currently available, in which the work in common is already done. An immediate goal is to provide an environment in which drivers can be analyzed with minimal effort, since there are many drivers which could be analyzed if a set of header files and libraries were prepared.

The ability to detect potential deadlocks will probably be added to warlock at some point, since warlock already does much of the work required to prepare for such analysis. Also, there may be ways to check for the proper use of semaphores and condition variables, at least in certain situations.

Warlock shows good potential for finding many – but not all – data races in multithreaded code written for Solaris. There is substantial opportunity to improve warlock's ability to understand code through the implementation of limited forms of data analysis. This will further improve the results of warlock's analysis, and reduce the need for conditional compilation.

References

[POSIX] POSIX P1003.4a Draft 5, IEEE.

[MH89] McDowell, C. E., and Helmbold, D. P. 1989. Debugging Concurrent Programs. ACM Computing Surveys 21, 4 (December), 593-622.

[Tay83] Taylor, R. N. 1983. A General-Purpose Algorithm for Analyzing Concurrent Programs. CACM 26, 5, 362-376.

[TO80] Taylor, R. N., and Osterweil, L. J. 1980. Anomaly Detection in Concurrent Software by Static Data Flow Analysis. IEEE Trans. Softw. Eng. SE-6, 3 (May), 265-278.

[YT88] Young, M., and Taylor, R. N. 1988. Combining Static Concurrency Analysis with Symbolic Execution. IEEE Trans. Softw. Eng. 14, 10 (October), 1499-1510.

[Pow91] Powell, M. L., Kleiman, S. R., Barton, S., Shah, D., Stein, D., and Weeks, M. 1991. SunOS Multi-thread Architecture. USENIX Conference Proceedings, January, 65-79.

[Kle92] Kleiman, S., Smaalders, B., Stein, D., and Shah, D. 1992. Writing Multithreaded Code in Solaris. IEEE CompCon Proceedings, February, 187-192.

Author Information

Nicholas Sterling is a member of the Kernel Tools Group at SunSoft, Inc., where tools are written to aid kernel development. There he has spent the last year developing warlock and helping OS developers use it. Nicholas holds a B.S. in Mathematics from the University of Arizona.

DUEL — A Very High-Level Debugging Language

*Michael Golan** & *David R. Hanson* — Princeton University

ABSTRACT

Most source-level debuggers accept expressions in the source language, e.g., C, and can print source-language values. This approach is usually justified on grounds that programmers need to know only one language. But the evaluation of source-language expressions or even statements is poorly suited for making non-trivial queries about the program state, e.g., "which elements of array `x[100]` are positive?" Duel departs from the conventional wisdom: It is a very high-level language designed specifically for source-level debugging of C programs. Duel expressions are a superset of C's and include "generators," which are expressions that can produce zero or more values and are inspired by Icon, APL, and LISP. For example, `x[..100] >? 0` displays the positive elements of `x` and their indices. Duel is implemented on top of `gdb` and adds one new command to evaluate Duel expressions and display their results. This paper describes Duel's semantics and syntax, gives examples of its use, and outlines its implementation. Duel is freely available and could be interfaced to other debuggers.

Introduction

Interactive source-level debuggers are now a standard part of nearly every programming environment. Most provide a rich suite of debugging facilities such as breakpoints, conditional breakpoints, watchpoints, stack traversals, etc., and many provide graphical user interfaces (GUIs) that use mouse actions and menus to invoke these facilities.

Despite these advances, the basics of debugging have changed little [8]. The basic debugging methodology is still to set breakpoints, run the program until a breakpoint is reached, and explore the program's state by displaying values of variables and data structures. GUIs make this exploration less tedious and more productive, but most are just a veneer over commands that print the values of expressions.

Programmers interact with most source-level debuggers in the language of the program being debugged, e.g., C debuggers accept and evaluate C expressions [9]. This choice is invariably justified on grounds that the programmer needs to know only one language, and that even small deviations would make a debugger unnecessarily hard to use [1, 8].

This paper investigates the contrary view: Programmers are best served by debugging languages that are more expressive and flexible than — and possibly different from — the program's source language. A concrete realization of this view is Duel, a very high-level language for debugging. Others have designed new debugger languages based, in part, on similar premises [7, 11, 12], and some recent work has focussed on semantic issues [3].

The overall "goal" of debugging is to search the program state for inconsistencies that manifest themselves as bugs. For example, questions like "which elements of array `x` are greater than `1`?," "how many nodes are in `tree`?," and "does list `L` contain two identical elements in its `value` fields?" typify the kinds of questions that can arise during state exploration.

Most debuggers can only print the values of expressions, which is of little help in answering complex queries. Some debuggers accept source-language statements or even procedures, but expressing these kinds of questions in languages such as C is tedious at best. For example, answering the query "does list `L` contain two identical elements in its `value` fields?" in C requires non-trivial code:

```
List *p, *q;
for (p = L; p; p = p->next)
   for (q = p; q; q = q->next)
      if (p->value == q->value)
         printf("%x %x contain %d\n",
            p, q, p->value);
```

This code also illustrates additional complexities, e.g., managing "debugger variables" (`p` and `q`).

*Supported in part by NSF Grant CCR92-00790.

Also, `printf` is a poor way to display the offending values, so the debugger must provide mechanisms for accessing its display functions. Even accessing these functions with special `printf` format codes forces programmers to use non-standard facilities when debugging.

Typical debugging queries are complex enough that experienced programmers write functions whose only use is to be called from the debugger. While undoubtedly useful, this methodology is invariably inadequate because programmers cannot anticipate all of the state exploration functions that might be needed.

Duel allows many state exploration queries to be expressed concisely, often as "one-liners" without additional variables or control constructs. Other capabilities include concise ways of printing parts of large data structures. Duel is derived mostly from C, Icon [6], a very high-level string-processing language, and, to a lesser extent, from APL and LISP. Duel is implemented on top of `gdb` [13], a traditional source-level debugger for C.

Design

Duel is an expression-oriented language in which expressions can return a sequence of values. Operators permit these sequences to be manipulated in novel ways to achieve the goal of concise state exploration. As a simple example, `x[0..9] >? 1` yields the elements of `x` that are greater than `1`, and `(x,y).a` yields the `a` field of `x` and of `y`. In the first example, the "`..`" operator produces the integers `0`, `1`,..., `9`. The C indexing operator is applied to `x` and each of those integers, producing the 0th through the 9th elements of array `x`. The "`>?`" operator compares its operands like C's "`>`" operator and returns the left one when the comparison is true. Each of `x`'s values is compared with `1`, and those greater than `1` are printed along with their indices.

Duel's semantics are modeled after Icon's; Duel's syntax, however, is quite different and is described below. Icon supports *generators* — expressions that can produce zero or more values — and goal-directed evaluation, which seeks the first "successful" result by trying all possible combinations of values generated by each subexpression. In contrast, Duel has no goal-directed evaluation; it produces all of the values of its generators, except for a few special operators. In many cases, expression evaluation in Icon and Duel is similar to evaluation in other languages, e.g., `x+y` adds `x` and `y`; there is only one possible value for each operand. The semantics and efficient implementation of generators are well documented [2, 4, 5].

Icon is only one of several languages that might be used as a basis for a high-level debugging language. The use of generators in Duel is more limited than in Icon, which is a complete, general-purpose, very high-level language. In addition to its generators, Duel includes some APL-style reduction operators and operators that expand data structure alá LISP.

Duel is designed primarily to debug C, but source-language expressions in most imperative languages could be extended with generators. Most of Duel operators could apply equally well to, e.g., Pascal, PL/I, FORTRAN, or C++.

Duel's semantics are more important than its syntax. Duel is used most effectively if its semantics are well understood, but the following two sections can be read in either order. Once the basics of generators are mastered, many of their uses become idiomatic.

Semantics

Duel's semantics are best described operationally using a C-like pseudo-language that mirrors the actual implementation. This pseudo-language omits punctuation, declarations, and error checking in the obvious ways. Duel users never write in this language; it is just a descriptive convenience. A similar approach has been used to describe Icon's semantics [10].

Duel evaluates an expression by traversing its *abstract syntax* tree (AST) recursively. All AST nodes have an `op` field, which identifies the node's operand, and a `kids` field, which is an a array of pointers to the operand nodes. Nodes for specific operators have additional fields, e.g., a node for a constant includes a `constant` field that holds the constant itself. The advantage of this notation is that it is independent of a specific concrete syntax. ASTs can be specified by a simple LISP-like notation, e.g., the AST for the expression `a*5 + *b` might be

```
(plus
   (multiply (name "a") (constant 5))
   (indirect (name "b"))
)
```

If all expressions returned only one value, `eval` would be a standard tree traversal:

```
Value eval(Node n) {
   Value u, v
   switch (n->op) {
   case CONSTANT:
      return n->constant
   case NAME:
      return fetch(n->name)
   case NEGATE, INDIRECT, ...:
      u = eval(n->kids[0])
      return apply(n->op, u)
   case PLUS, MINUS, MULTIPLY, ...:
      u = eval(n->kids[0])
      v = eval(n->kids[1])
      return apply(n->op, u, v)
   ...
   }
}
```

eval switches on the operator, recursively evaluates the operands, if necessary, and calls **apply** to evaluate a specific operator. **fetch** retrieves the value of the variable named in the **NAME** node's **name** field. **Value** denotes a type that encapsulates all values.

Duel expressions can produce more than one value, e.g., **(1..3)+(5,9)** prints **6 10 7 11 8 12**. **(1..3)** produces **1**, **2**, and **3**, and **(5,9)** produces each of its "alternatives," **5** and **9**. The "+" sums all possible combinations of these values.

This feature complicates only **eval**, not the code for each individual operator, i.e., instead of changing all of the operators to take lists of values, **eval** manages the multiple values. Each call to **eval** produces one of the values. To implement this version of **eval**, state information is added to each node, and a distinguished value, **NOVALUE**, signals the end of a sequence of values. The **state** field of a node is a non-negative integer that indicates the progress of the evaluation of that node. **state** begins at **0** and is changed to **1** before the first value is returned to indicate that subsequent calls to **eval** for this node will return additional values. **goto** statements are used in the code below to emphasize this flow of control. After **NOVALUE** is returned, the next call to **eval** re-evaluates the node. The field **value** is a temporary value that must be saved between successive calls to **eval**. For example, this version of **eval** handles constants and most of the binary operators as follows; the line numbers are for explanatory purposes and are not part of the code.

```
Value eval(Node n) {
   switch (n->op) {
   case CONSTANT:
      if (n->state == 0) {
         n->state = 1
         return n->constant
      } else {
         n->state = 0
         return NOVALUE
      }
   ...
1  case PLUS, MINUS, MULTIPLY, ...:
2        if (n->state == 1) goto bin1
3  bin0: n->state = 0
4        n->value = eval(n->kids[0])
5        if (n->value == NOVALUE)
6           return NOVALUE
7        n->state = 1
8  bin1: u = eval(n->kids[1])
9        if (u == NOVALUE) goto bin0
10       v = apply(n->op, n->value, u)
11       return v
   ...
   }
}
```

To understand this code, consider the evaluation of the addition in the expression **(1..3)+(5,9)**, which has the AST

```
(plus (to 1 3) (alternate 5 9))
```

When **eval** is called with the **plus** node, control lands at line 4 above and **eval** is called with the **(to 1 3)** node. This recursive invocation of **eval** returns **1**, which is saved in the **plus** node's **value** field. The **plus** node's **state** is reset to **1**, and control ultimately lands at line 8. This second call to **eval** on **(alternate 5 9)** returns **5**, **apply** computes the sum, **6**, which is the return value from the top-level **eval**.

Duel's top-level evaluation command "drives" its expression argument and prints *all* of its values. So, **eval** is called again with the **plus** node as its argument. This time, the **plus** node's **state** is **1**, so control lands at line 8, and **eval** is called recursively for the next value from the node **(alternate 5 9)**. This call returns **9**, which causes the top-level call to **eval** to return **10**, which is printed.

Ultimately, the call to **eval** in line 8 returns **NOVALUE**, control passes to line 3, the **plus** node's **state** is reset to **0**, and line 4 calls **eval** for the next value from **(to 1 3)**. This call returns **2**, the **state** is reset to **1** again, and the whole process of re-evaluating **(alternate 5 9)** begins anew and produces **5** again.

This process continues until all of the values from **plus**'s first operand have been produced, which occurs when the call to **eval** in line 4 returns **NOVALUE**. Finally, line 6 announces that the entire **plus** expression has produced all of its values. If

`eval` is called again with this `plus`, the entire evaluation process starts over because `state` has been reset to `0`.

Each of Duel's generators has a similar implementation scheme. This scheme simulates coroutines (which are similar to, but pre-date, non-preemptive threads).

Managing the `state` and `value` correctly for each generator according to its semantics is straightforward, but tedious. The semantics are conveyed equally well by assuming that `eval` is a coroutine in which the values of local variables are saved across calls, and that the statement `yield` e returns e and preserves enough information for the computation to resume after the `yield` statement. (`alternate` e_1 e_2) produces all of the values of e_1 followed by the values of e_2. Its detailed implementation is

```
case ALTERNATE:
      if (n->state == 1) goto alt1
      u = eval(n->kids[0])
      if (u != NOVALUE)
         return u
      n->state = 1
alt1: v = eval(n->kids[1])
      if (v != NOVALUE)
          return v
      n->state = 0
      return NOVALUE
```

The simplified code is

```
case ALTERNATE:
   while ((u = eval(n->kids[0])) != NOVALUE)
      yield u
   while ((v = eval(n->kids[1])) != NOVALUE)
      yield v
   return NOVALUE
```

Declarations, explicit comparisons with `NOVALUE`, and the final "`return NOVALUE`" are omitted when the meaning is clear, e.g.,

```
case ALTERNATE:
   while (u = eval(n->kids[0]))
      yield u
   while (v = eval(n->kids[1]))
      yield v
```

Most of the unary operators are defined by

```
case NEGATE, INDIRECT, ...:
   while (u = eval(n->kids[0]))
      yield apply(n->op, u)
```

Specific Operators

The generator (`to` e_1 e_2) produces the integers from e_1 to e_2 inclusive. The semantics of `to` are defined by

```
case TO:
   while (u = eval(n->kids[0]))
      while (v = eval(n->kids[1]))
         for (i = u; i <= v; i++)
            yield i
```

As suggested by this code, `to`'s operands can be generators, e.g.,

```
(to (alternate 1 5) (alternate 5 10))
```

produces

```
1 2 3 4 5
1 2 3 4 5 6 7 8 9 10
5
5 6 7 8 9 10
```

Some operators return one or *no* value. Duel's comparisons produce their first operand if the condition is true and nothing otherwise, e.g., (`ifgt` e_1 e_2) produces e_1 only if e_1 is greater than e_2. The implementation is

```
case IFGT, IFGE, IFLE, IFLT, IFEQ, IFNE:
   while (u = eval(n->kids[0]))
      while (v = eval(n->kids[1]))
         if (w = apply(n->op, u, v))
            yield w
```

These semantics admit generators as operands, so an expression like

```
(ifgt
   (index (name "x") (to 0 99))
   (constant 0)
)
```

produces the positive elements of the array `x[100]`.

The operators that correspond to the C operators `&&` and `||` can be problematic because, with generator operands, their semantics are nonintuitive. The semantics of `andand` illustrate the problem.

```
case ANDAND:
   while (u = eval(n->kids[0]))
      if (u != 0)
         while (v = eval(n->kids[1]))
            yield v
```

e_1 `&&` e_2 produces *all* of the values of e_2 for each non-zero value produced by e_1. When e_1 and e_2 are single-value expressions, these semantics are equivalent to C's.

The operator (`if` e_1 e_2 e_3) evaluates e_1; for each non-zero value of e_1, it produces all of the values of e_2, and for each zero value of e_1, it produces all of the values of e_3.

```
case IF:
   while (u = eval(n->kids[0]))
      if (u != 0)
         while (v = eval(n->kids[1]))
            yield v
      else
         while (v = eval(n->kids[2]))
            yield v
```

A sequence of expressions, (**sequence** e_1 e_2) evaluates e_1 but discards its values, and then produces the values of e_2.

```
case SEQUENCE:
   while (u = eval(n->kids[0]))
      ;
   while (v = eval(n->kids[1]))
      yield v
```

(**imply** e_1 e_2) is similar, but produces e_2's values for each value of e_1.

```
case IMPLY:
   while (u = eval(n->kids[0]))
      while (v = eval(n->kids[1]))
         yield v
```

Finally, (**while** e_1 e_2) produces e_2 only if *all* of the values of e_1 are non-zero:

```
case WHILE:
   for (;;) {
      while (u = eval(n->kids[0]))
         if (u == 0)
            return NOVALUE
      while (v = eval(n->kids[1]))
         yield v
   }
```

These semantics are equivalent to C's when e_1 is a single-value expression. Notice that once e_2 has produced all of its values, **while** starts over again. For example,

```
(while (index (name "x") (to 0 99)) ...)
```

produces "..." as long as all of the elements of **x** are non-zero.

Some operators manipulate value sequences instead of values themselves. For example, (**select** e_1 e_2) produces the elements of e_2 given by the integers in e_1. The implementation is described by

```
case SELECT:
   while (v = eval(n->kids[0])) {
      n->kids[1]->state = 0
      for (i = 0; i < v; i++)
         u = eval(n->kids[1])
      yield u
   }
```

Notice that e_2's **state** is reset so that it starts anew for each value of e_1. The actual implementation of **select** avoids the re-evaluation of e_2 when possible.

Several "reduction" operators reduce a sequence of values to one value, e.g., (**count** e) returns the number of values produced by e, (**sum** e) sums the values produced by e and (**equality** e_1 e_2) returns **1** if the values produced by e_1 are equal to those produced by e_2 and **0** otherwise.

Duel's evaluation mechanism also applies to calls to functions in the target program. If any of the arguments are generators, the function is called repeatedly for all combinations of values, e.g.,

```
printf("%d %d, ", (3,4), 5..7)
```

prints

```
3 5, 3 6, 3 7, 4 5, 4 6, 4 7,
```

Aliases

As suggested above, (**name** "x") fetches the value of the variable x. x can be a variable in the target program or an *alias*. Aliases are created by (**define** a e), which defines a to be an alias for e. If e is an lvalue, so is a, e.g., after (**define b x[5]**), changing **b** changes **x[5]**. If e is a generator, a is aliased to each value in turn and **define** returns those values.

```
case DEFINE:
   while (u = eval(n->kids[1]) {
      alias(n->name, u)
      yield u
   }
```

The operator (**with** e_1 e_2) evaluates e_2 in the "scope" of e_1. When e_1 is a structure, "opening the scope" of e_1 makes the fields visible as ordinary identifiers. Names in e_2 refer to the appropriate fields in e_1; for example, if **x** and **y** are instances of structures with a field **f**,

```
(with
   (alternate (name "x") (name "y"))
   (alternate (name "f") (name "g"))
)
```

generates **x.f**, **x.g**, **y.f**, and **y.g**. The semantics of **with** are defined as follows.

```
case WITH:
   while (u = eval(n->kids[0]) {
      push(u)
      while (v = eval(n->kids[1])
         yield v
      pop()
   }
```

The **push** and **pop** functions manipulate the name-resolution stack used by **fetch**. Also, the special name "_" in e_2 refers to the value of e_1.

The **with** operator is used by other operators to traverse data structures. **(dfs** e_1 e_2**)** "expands" the data structure e_1 using e_2 to indicate the traversal path as follows. Unvisited nodes are kept in a stack. At each step, the top of the stack, X, is popped, the non-null values generated by **(with** $*X$ e_2**)** are pushed onto the stack, and **dfs** yields X. This process continues until the stack is empty. In the semantics below, **stack** and **unstack** manipulate n's traversal stack and **push** and **pop** manipulate the name-resolution stack described above.

```
case DFS:
   while (u = eval(n->kids[0]) {
      stack(n, u)
      while (v = unstack(n)) {
         push(v)
         while (w = eval(n->kids[1])
            stack(n, w)
         pop()
         yield v
      }
   }
```

If **head** is a pointer to a linked list in which nodes are linked via **next** fields,

```
(dfs (name "head") (name "next"))
```

generates the elements of the list. Likewise,

```
(dfs
   (name "root")
   (alternate (name "left") (name "right"))
)
```

generates the nodes in a binary tree in preorder. (The actual implementation stacks the values of e_2 in reverse order so that the nodes are visited in the expected order.) Other operators do similar expansions with different orderings, e.g., breadth-first search.

Syntax

Duel uses an extended C-like concrete syntax to specify the semantics described above. Duel accepts expressions, compiles them into ASTs, evaluates them, and prints the resulting values. Expressions include all of the C operators with the expected semantics except for ",", and C's scope rules apply. Control structures, like **for**, **if**, etc. are cast as expressions, not statements, much as in Icon. Finally, there are numerous Duel-specific operators that specify the generators described above.

In the absence of generators, Duel expressions are essentially equivalent to a debugger's "print" command, e.g.,

```
gdb> print 1 + (double)3/2
2.500
gdb> duel 1 + (double)3/2
2.500
```

As this example suggests, Duel is an extended version of **gdb**; the **duel** command is similar to **gdb**'s **print** command, except that the **duel** command drives its expression argument and prints all of its values, e.g.,

```
gdb> duel (1,2,5)*4+(10,200)
14 204 18 208 30 220
gdb> duel (3,11)+(5..7)
8 9 10 16 17 18
```

The comma operator is the concrete syntax for the **alternate** operator described in the previous section; e_1**,** e_2 produces all of e_1's values followed by e_2's values. The operator ".." specifies the **to** operator; e_1**..**e_2 produces the integers from e_1 to e_2 inclusive.

Duel treats lvalues and rvalues as in C. For example, suppose that **hash** is defined by the declaration

```
struct symbol {
   char *name;
   int scope;
   struct symbol *next;
} *hash[1024];
```

which is a typical representation for symbol tables in compilers. **hash** is an array of pointers to lists of **symbol** structures, the lists are threaded through the **next** fields, and the **symbol**s are in decreasing order of the **scope** fields. The command

```
gdb> duel hash[0..1023]->scope = 0 ;
```

clears the **scope** field of the first **symbol** on each list. **hash[0..1023]** produces lvalues; the semantics of C's assignment are unchanged. This example produces no output; the terminating semicolon causes the expression to be evaluated for side effects only.

The operator **>?** specifies the operator **ifgt**; e_1 **>?** e_2 returns e_1 if e_1 is greater than e_2 and nothing otherwise. This operator and the similar ones for the other comparisons can be used with other generators like ".." to search for specific values. For example,

```
gdb> duel x[1..4,8,12..50] >? 5 <? 10
x[3] = 7
x[18] = 9
x[47] = 6
```

searches portions of **x** for values that are between **5** and **10**. Duel's output includes symbolic expressions that suggest the derivation of the values printed. Thus, the output from the search shows not only the desired values, but also pinpoints the elements of **x** that hold those values. (The examples at the beginning of this section omitted the symbolic output.) The command **x[1..4,8,12..50] ==? (6..9)** is another formulation of the same search.

Duel also supports the C operators, **==**, etc., but their semantics are as in C, e.g.,

```
gdb> duel x[1..3] == 7
x[1]==7 = 0
x[2]==7 = 0
x[3]==7 = 1
```

prints *all* of the indices and values of **x**.

The unary expression "*..e*" is shorthand for **0..***e*-**1** and is useful for indexing arrays. For example,

```
gdb> duel (hash[..1024] !=? 0)->scope >? 5
hash[42]->scope = 7
hash[529]->scope = 8
```

prints the elements in **hash** that have a **scope** value greater than **5**.

This example illustrates the crux of the problem in designing Duel's syntax. It must necessarily be a superset of C, but the wealth of operators quickly overwhelms the vocabulary that permits a readable notation for them.

In programming language design, readability is important because programs are read more than written, e.g., in debugging and maintenance. Duel expressions, however, are ephemeral; they exist only long enough to be executed once. They are written once and read at most once. Duel's syntax is designed to facilitate, on-the-fly, left-to-right composition, e.g., **hash[..1024]** specifies all of the lists, **!=? 0** specifies those that are non-null, **->scope** specifies the **scope** fields of just those elements, and **>? 5** limits the output to the desired elements. While these kinds of expressions appear cryptic initially, they become idiomatic with use. At the very least, Duel expressions are more compact than the equivalent C code, e.g., the C (and Duel) code for the search just described is

```
gdb> duel int i;
   for (i = 0; i < 1024; i++)
      if (hash[i] != 0)
         if (hash[i]->scope > 5)
            printf("hash[%d]->scope = %d\n",
               i, hash[i]->scope);
```

Duel declarations, e.g., **int i**, establishes aliases to newly allocated target locations.

Duel accepts most of C, and C and Duel expressions can be intermixed freely. For example, the following Duel lines all print the same **scope** fields as the search of **hash** described above.

```
gdb> duel int i; for (i = 0; i < 1024; i++)
      if (hash[i] && hash[i]->scope > 5)
         hash[i]->scope

gdb> duel int i; for (i = 0; i < 1024; i++)
      if (hash[i]) hash[i]->scope >? 5

gdb> duel int i; for (i = 0; i < 1024; i++)
      (hash[i] !=? 0)->scope >? 5
```

As suggested by its semantics, Duel's **if** is an expression, e.g.,

```
gdb> duel for (i = 0; i < 9; i++)
            4 + if (i%3==0) i*5
4+i*5 = 4
4+i*5 = 19
4+i*5 = 34
```

The appearance of "**i**" instead of its value in this example illustrates a potentially unappealing side effect of Duel's symbolic display algorithm. The algorithm substitutes the actual value only for generators; other expressions are displayed as entered. Enclosing an expression in braces overrides the default display for that expression and causes its value to be displayed, e.g.,

```
gdb> duel for (i = 0; i < 9; i++)
            4 + if (i%3 == 0) {i}*5
4+0*5 = 4
4+3*5 = 19
4+6*5 = 34
```

The semicolon specifies the **sequence** operator, which evaluates but discards its left operand, and returns its right operand, e.g.,

```
gdb> duel i := 1..3; i + 4
i+4 = 7
```

imply is specified by **=>**; e_1**=>**e_2 produces e_2's values for each of e_1's values, e.g.,

```
gdb> duel i := 1..3 => {i} + 4
1+4 = 5
2+4 = 6
3+4 = 7
```

The operator a := e defines a to be an alias for e, which may be either an lvalue or an rvalue, e.g.,

```
x := hash[..1024] !=? 0 =>
   y := x->scope => y = 0
```

clears the **scope** fields of the **symbols** in **hash**. **x** is an alias for each element of **hash** and **y** is an alias for each **scope** field.

The operators "." and -> specify Duel's **with** operator; as in C, "." applies to structures and -> applies to pointers to structures. In both e_1.e_2 and e_1->e_2, e_2 is evaluated within the scope of e_1. For example, alternation can specify several fields of a structure:

```
gdb> duel hash[1,9]->(scope,name)
hash[1]->scope = 3
hash[1]->name = "x"
hash[9]->scope = 2
hash[9]->name = "abc"
```

The "." and -> operators are quite general, e.g.,

```
x := hash[..1024] !=? 0 =>
   x->(if (scope > 5) name)
```

prints the **name** field of the elements in **hash** that have a **scope** greater than 5. References to "_" refer to **with**'s operand, which helps eliminate temporaries like **x** in the example above; for instance, the example above can be done by

```
hash[..1024]->(if (_ && scope > 5) name)
```

Using "_" instead of an alias often produces more informative output. For example

```
gdb> duel y := x[..10] =>
             if (y < 0 || y > 100) y
y = -9
y = 120

gdb> duel x[..10].if (_ < 0 || _ > 100) _
x[3] = -9
x[8] = 120
```

The first command uses an alias for each element of **x** and prints those elements that are less than 0 or greater than 100. The output displays the name of the alias, not the elements of **x**. The "_" stands for the value itself, an element of **x** in this example, so the output of the second command displays the specific elements of **x** that are generated. The same effect can be achieved with aliases but requires another temporary:

```
y := x[j := ..10] =>
   if (y < 0 || y > 100) x[{j}]
```

The operator --> specifies the **dfs** node described in the previous section. e_1-->e_2 produces the values from the data structure given by e_1 using e_2 to specify the traversal. For example, if **head** points to a linked list of structures threaded through **next** fields, **head-->next** produces the elements of the list, i.e., it produces **head**, **head->next**, **head->next->next**, etc., until a **NULL** pointer or an invalid pointer terminates the sequence. So,

```
gdb> duel hash[0]-->next->scope
hash[0]->scope = 4
hash[0]->next->scope = 3
hash[0]->next->next->scope = 2
hash[0]->next->next->next->scope = 1
```

prints the **scope** fields of the list emanating from **hash[0]**. Specifying **hash[..1024]** would print the **scope** fields for all of the **symbols** in the table.

The expression

```
L-->next->(value ==? next-->next->value)
```

answers the Introduction's query about list **L** containing identical elements in its **value** fields. **L-->next** generates each element in **L**, the **value** field of which is compared to the **value** fields of each of the succeeding elements generated by **next-->next**. Compare this compact expression with the C code given in the Introduction. The longer C code hides a bug: the initialization of the inner **for** loop should be **q = p->next**.

Suppose a binary tree is composed of nodes in which each node includes an integer **key** and **left** and **right** fields that point to the subtrees, and that **root** is the head of the binary tree specified in preorder as **(9 (3 (4) (5)) (12))**. The keys in the entire tree are printed by

```
gdb> duel root-->(left,right)->key
root->key = 9
root->left->key = 3
root->left->left->key = 4
root->left->right->key = 5
root->right->key = 12
```

and the path to the node holding 5 is printed by

```
gdb> duel root-->(if (key < 5) left
            else if (key > 5) right)->key
root->key = 9
root->left->key = 3
root->left->right->key = 5
```

Another, more complex, example is

```
gdb> duel hash[..1024]-->next->
           if (next) scope <? next->scope
hash[287]-->next[[8]]->scope = 5
```

`hash[..1024]-->next` produces all of the nodes on all of the lists in `hash`. The `if` expression returns a `scope` field only if it is less that the `scope` field of the next element on the list. Thus, this command verifies that the `symbols` in each list are sorted in decreasing order of `scope`, as expected. This output displays the error. The symbolic display algorithm automatically prints occurrences of $\texttt{->}a\texttt{->}a$ as $\texttt{-->}a\texttt{[[2]]}$, etc.

The `select` operator is specified by $e_1\texttt{[[}e_2\texttt{]]}$ and produces the values from e_1 as specified by the values in e_2. For example,

```
gdb> duel ((1..9)*(1..9))[[52,74]]
6*8 = 48
9*3 = 27
gdb> duel hash[287]-->next[[7..10]]->scope
hash[287]-->next[[7]]->scope = 6
hash[287]-->next[[8]]->scope = 5
hash[287]-->next[[9]]->scope = 9
hash[287]-->next[[10]]->scope = 3
```

The reduction operators help summarize the contents of data structures, e.g., `count` is specified by `#/`e and counts the number of values produced by e:

```
gdb> duel #/(root-->(left,right)->key)
5
```

The operator unary $e\texttt{\#}n$ produces the values of e and arranges for n to be an alias for the index of each value in e. Thus, if `L` is the list mentioned in the Introduction and its 4th and 9th nodes each contain 27 the following command displays the duplication.

```
gdb> duel L-->next#i->value ==?
          L-->next#j->value =>
            if (i < j) L-->next[[i,j]]->value
L-->next[[4]]->value = 27
L-->next[[9]]->value = 27
```

The expression $e\texttt{@}n$ produces the values of e until $e.n$ is non-zero. For example, is `s` is a pointer to a character, `s[0..999]@(_=='\0')` produces `s[0]`, `s[1]`, ...up to but not including the terminating null character. Also, n can be a constant, in which case the expression produces the values of e up to the first one that equals n. "e`..`" generates an essentially infinite sequence of integers beginning at e, so `argv[0..]@0` generates the strings in `argv`.

Implementation

Duel is designed to be implemented as an add-on to existing debuggers. Currently, it is interfaced only to `gdb`, but Duel is not derived in any way from `gdb`. Duel works wherever `gdb` does and can be used with `emacs` and other debugger front ends.

Duel's `yacc`-based parser and the hand-written lexer accept a Duel expression and compile it into an abstract syntax tree. The nodes in the AST correspond to the primitive operators described above.

Evaluation is implemented by `duel_eval`, which is the actual function corresponding to the abstract function `eval` use to describe the semantics. `duel_eval`'s code for most of the operators is equivalent to the pseudo-code that describes their semantics.

`duel_eval` and its associated functions are about 400 lines of C. Related functions, which manipulate search stacks, aliases, etc., are another 300 lines, and the operator application functions, including `Value` manipulations, consist of about another 1200 lines. The graph-expansion operators, e.g., `-->`, are implemented as described above, but the current implementation does not handle cycles.

As for other very high-level languages, type checking must be done during evaluation, not during compilation. For example, in `(x,y).a`, `x` and `y` can each have any structure type with a field named `a`. Consequently, the ASTs are decorated with symbolic values, like `a`, instead of pointers to symbol-table entries as in most compilers.

While evaluation-time type checking is flexible, it costs time. For example, most of the time in evaluating `1..100+i` goes to the `100` lookups of `i`. The current implementation of `duel_eval` is flexible to allow experiments with different semantics and syntax, but more efficient implementations of generators are possible [14]. The evaluation time for most Duel expressions is negligible. For example, `x[..10000] >? 0` compiles and executes in about 5 seconds on a DECStation 5000. A faster implementation would be required if Duel expressions were used in watchpoints and conditional breakpoints. For many Duel expressions, run-time type checking and symbol lookup could be done at compile time using type-inference techniques.

The "values" produced during evaluation have a type, an actual value, and a symbolic value. The actual value is a value of a primitive C type or an lvalue, which is a pointer to target data. The symbolic value is a symbolic expression (i.e., a legal Duel expression) that indicates how the value was computed. The symbolic value of a variable

is its name; for most binary operators $\times$, the symbolic value is $a \times b$ where a and b are the symbolic values of the operands. Some operators have symbolic values that relate better to the computation at hand, e.g., `a..b`'s symbolic value is the current iteration value. Symbolic values assist in the display of results as well as errors: The offending operand's symbolic value is printed, e.g., the expression `ptr[..99]->val` might produce

```
Illegal memory reference in x of x->y:
   ptr[48] = lvalue 0x16820.
```

The symbolic value of an expression is computed at the same time that the expression is evaluated, e.g., in `x[1+2]` the strings `"1+2"` and `"x"` are combined to produce `"x[1+2]"` at the same time that the lvalue `&x+3` is computed. In most cases, the computation of the symbolic value is more expensive than computing the result. Furthermore, many of the symbolic computations are unnecessary, because they are never printed, e.g., in `x[..1000] !=? 0`, the symbolic expression `x[i]` is computed 1000 times, even though it might be printed only once. This kind of overhead is noticeable in complex queries and would need to be eliminated if such queries were used in watchpoints and conditional breakpoints.

Duel's interface to a debugger is a two-way interface and is intentionally narrow to simplify connecting it to a debugger. Duel duplicates some debugger capabilities in order to reduce its dependence on specific debuggers. For example, Duel contains its own type and value representations and its own implementation of the C operators.

The only new **gdb** command, **duel** *expr*, accepts a Duel expression and passes it as a string to Duel's single entry point. The only modification to **gdb** was the change to one line to allow # in commands (# starts a comment in **gdb**; Duel uses ##). A single module contains the interface code between **Duel** and **gdb**. This module is about 400 lines of C broken down as follows.

30	**duel** command
100	converting between **gdb** and Duel types
100	symbol-table functions
70	accessing the target's address space
100	miscellaneous

Duel calls functions to allocate memory, read and write the target's data space, and to determine the types and addresses of target symbols. It does not call any **gdb** functions.

Duel's debugger interface consists of the following functions.

duel_get_target_bytes
duel_put_target_bytes:
: copies **n** bytes to/from a target address.

duel_alloc_target_space:
: allocates **n** bytes in the target space.

duel_call_target_func:
: calls a function in the target.

duel_get_target_variable:
: returns value/type information for a symbol.

duel_get_target_typedef/struct/union/enum:
: returns type information for a symbol.

Except for type and value conversions, most of these functions simply call **gdb** equivalents. Only a few other miscellaneous functions are needed, e.g., to find the number of active frames, to retrieve bit fields in a machine-dependent way, etc.

Duel has been "ported" only from **gdb** 4.2 to **gdb** 4.6 on both SUN and DEC workstations. It has also been tested as a stand-alone program under MS-DOS. The change in **gdb** versions required modifications to only 4 lines of code in the interface module because internal **gdb** structures changed.

Discussion

Initial experience with Duel suggests that its generators are an effective way to explore program state. Once the initial implementation was working, it was used to probe both itself and **gdb**. This exploration not only uncovered bugs, but helped to understand the inner workings of **gdb**, which was necessary for designing and implementing the Duel–debugger interface.

As expected, Duel's syntax remains a potential hurdle. Understanding the semantics independently of the syntax helps, but programmers must interact with the debugger at some syntactic level, so Duel's syntax continues to evolve. Alternatives are also under consideration. For example, some database query languages use a visual programming approach to composing queries. Duel might benefit from a similar approach, especially if it maintained a history so that common, program-specific queries could be made by simply pointing and clicking. Allowing such history lists to be edited might also help.

Currently, Duel expressions can refer only to program variables. For example, displaying the local **x** in *all* of the currently active stack frames for the function that declares **x** is tedious to do with most debuggers. Mechanisms for exploring such "unnamed" portions of the program state would be useful and are under investigation. Duel would also

be useful in other traditional debugging facilities, e.g., watchpoints and conditional breakpoints.

Duel's linguistic framework might apply to other programming environment facilities that rely on program state exploration. Assertions, for example, make claims about the state at various points in a program. Complex assertions, e.g., "`x[0]` through `x[n]` are positive," often need non-trivial code to compute the assertion outcome. Annotating programs with assertions written in a Duel-like language might simplify making these kinds of assertions and encourage their use.

Availability

Duel is public-domain software. It is available for anonymous `ftp` from `ftp.cs.princeton.edu` in the directory `pub/duel`.

References

[1] B. Beander. VAX DEBUG: An interactive, symbolic, multilingual debugger. *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, SIGPLAN Notices*, 18(8):173–179, Aug. 1983.

[2] T. A. Budd. An implementation of generators in C. *Journal of Computer Languages*, 7(2):69–82, Mar. 1982.

[3] R. H. Crawford, R. A. Olsson, W. W. Ho, and C. E. Wee. Semantic issues in the design of languages for debugging. In *Proceedings of the International Conference on Computer Languages*, pages 252–261, Oakland, CA, Apr. 1992.

[4] R. E. Griswold. The evaluation of expressions in Icon. *ACM Transactions on Programming Languages and Systems*, 4(4):563–584, Oct. 1982.

[5] R. E. Griswold and M. T. Griswold. *The Implementation of the Icon Programming Language.* Princeton University Press, Princeton, NJ, 1986.

[6] R. E. Griswold and M. T. Griswold. *The Icon Programming Language.* Prentice Hall, Englewood Cliffs, NJ, second edition, 1990.

[7] M. S. Johnson. The design of a high-level, language independent symbolic debugging system. In *Proceedings of the ACM Annual Conference*, pages 315–322, Seattle, WA, Oct. 1977.

[8] M. S. Johnson. *The Design and Implementation of a Run-Time Analysis and Interactive Debugging Environment.* PhD thesis, The University of British Columbia, Aug. 1978.

[9] M. A. Linton. The evolution of Dbx. In *Proceedings of the Summer USENIX Technical Conference*, pages 211–220, Anaheim, CA, June 1990.

[10] J. O'Bagy and R. E. Griswold. A recursive interpreter for the Icon programming language. *Proceedings of the SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques, SIGPLAN Notices*, 22(7):138–149, July 1987.

[11] R. A. Olsson, R. H. Crawford, and W. W. Ho. Dalek: A GNU, improved programmable debugger. In *Proceedings of the Summer USENIX Technical Conference*, pages 221–231, Anaheim, CA, June 1990.

[12] R. A. Olsson, R. H. Crawford, W. W. Ho, and C. E. Wee. Sequential debugging at a high level of abstraction. *IEEE Software*, 8(3):27–35, May 1991.

[13] R. M. Stallman and R. H. Pesch. Using GDB: A guide to the GNU source-level debugger, GDB version 4.0. Technical report, Free Software Foundation, Cambridge, MA, 1991.

[14] K. Walker and R. E. Griswold. An optimizing compiler for the Icon programming language. *Software—Practice & Experience*, 22(8):637–657, Aug. 1992.

Author Information

Michael Golan is a graduate student in the PhD program in Computer Science at Princeton University. His research interests include programming environments and software engineering. He can be reached via US mail at Dept. of Computer Science, Princeton University, 35 Olden St., Princeton, NJ 08544 and via electronic mail at `mg@cs.princeton.edu`.

David R. Hanson received his PhD in Computer Science from the University of Arizona in 1976. He has held faculty positions at Yale and the University of Arizona and was Dept. Head at Arizona from 1981–86. His visiting appointments include the University of Utah, the Institute for Defense Analyses, Adobe Systems, and Digital's Systems Research Center. In 1986, he joined Princeton University, where he is currently Professor of Computer Science. He was co-editor of *Software—Practice & Experience* from 1980–88 and continues to serve on its editorial board. He can be reached via US mail at Dept. of Computer Science, Princeton University, 35 Olden St., Princeton, NJ 08544 and via electronic mail at `drh@cs.princeton.edu`.

The San Diego "Zoo": A multicomputer stress test suite

Chris Peak – Locus Computing Corporation, San Diego

ABSTRACT

This paper describes a suite of stress tests for the OSF/1 AD TNC operating system running on Intel's Paragon XP/S and iPSC/860 Hypercube multicomputers or on networked AT386 machines. These tests were written to exercise the distributed process management features of this OS, but to do so using unsophisticated user-level programs as much as possible. In particular, much use is made of Korn shell scripts supplemented by a minimal number of standard TNC user commands.

The zoomorphic behavior of these tests – involving spontaneous movement, sleeping, eating (cpu time) – suggested their animal names. Coincidentally, the Locus office is located in San Diego, so the test suite became dubbed the San Diego "Zoo" and soon took on a life of its own. And what could be more stressful than life itself?

Note that despite the whimsical tone of this paper, the subject and approach outlined here are quite real and applicable to any distributed process environment with remote execution and process migration capabilities.

Background and Objectives

OSF/1 AD TNC was developed by Locus Computing Corporation in conjunction with OSF's Research Institute for the Supercomputer Systems Division of Intel Corporation. This OS is described in more detail by an associated paper. The system was designed to exploit the potential of massively parallel processing (MPP) architectures from the comfort of a familiar POSIX-compliant environment.

TNC (Transparent Network Computing) provides distributed process management and remote processing capabilities that are transparent to the user program. Standard OSF/1 binaries run unchanged under TNC yet can be migrated transparently between nodes of the host multicomputer – either under the direction of a load-leveler daemon or other TNC user commands. Specialized programs can use additional remote processing primitives – in particular, `rfork()`, `rexec()` and `migrate()` – to further exploit the MPP resources.

OSF/1 AD TNC was tested with the usual battery of conformance tests: VSX and VSE running on a single node system, plus VSTNC – a specially-written suite testing the functions of TNC. AIM-III was used to stress a single node system, but a new stress suite was required to stress TNC's distributed process management in multinode configurations.

Zoo became the nickname for tests developed to meet this objective. A series of TNC-aware tests are scattered in a largely random pattern throughout the nodes of the host system. Most tests migrate randomly between nodes, further distributing stress. As one test completes, another is spawned to take its place. Hence, a constant overall workload can be maintained over a given time period. Heavy demands can be placed on the remote processing primitives of TNC and its ability to manage process relationships over node boundaries – for example, to deliver inter-process signals regardless of physical location of any affected process. Moreover, the underlying Mach microkernel primitives are stressed indirectly by the suite.

Note that the automatic load-leveling capability of TNC was itself unsuitable to stress the system. For, although *exercising* TNC's distributed process management, the load-leveling algorithms are specifically devised to *minimize* stress.

Zoo tests are predominantly simple Korn Shell scripts. They are able to exploit TNC capabilities through a small number (3) of TNC user commands which are described in the next section. Additionally, Zoo contains a few more exotic beasts which have been bred for specific purposes.

Habitat

OSF/1 AD TNC is a POSIX-compliant operating system which itself runs as a server task under the NORMA-Mach3.0 microkernel.

The AD TNC server runs OSF/1 binaries unmodified; it is also BSD4.3 compatible - indeed, the copy of ksh used by this test suite for AT 386 machines was originally built for a Mach2.5 BSD integrated kernel system. Three TNC user command programs are required – **onnode**, **node_self** and **kill3** – which are described individually after a brief introduction is given of the basic TNC remote processing primitives.

OSF/1 AD TNC Environment

OSF/1 AD TNC is a development of the OSF/1 MK system, a server-based version of OSF/1 IK integrated kernel. OSF/1 IK is a monolithic system wherein Unix semantics are provided by a layer of code integrated with – occupying the same address space as – the Mach microkernel. OSF/1 MK moves the Unix functions out of micro-kernel space and into a server task. AD further separates file service and process management functions and enables user processes to be statically distributed over multicomputer nodes. User processes communicate with server tasks using NORMA-Mach internode IPC messaging. TNC adds the ability to distribute user processes dynamically and transparently throughout nodes. Refer to the paper entitled "An OSF/1 Unix for Massively Parallel Processor Systems" for an in-depth look at the architecture of OSF/1 AD TNC.

TNC partitions PID space by allocating system-wide PIDs in which the most significant 16 bits indicates the origin node number of a process. Although a process may move between nodes, it retains the PID allocated when it was created (forked). TNC ensures that the origin node of a process tracks the execution node: hence, a process may be located by its PID directly or indirectly through its origin node.

Under TNC, processes move between nodes directly under program control using a small set of supplementary system calls. Additionally, they may be distributed under the control of a load-leveling daemon – which exploits the TNC SIGMIGRATE signal to cause a process movement.

TNC System Calls

The following additional system calls are provided by TNC:

`rfork()`

Taking an additional node parameter over the conventional `fork()` system call, `rfork()` forks a child copy of the caller onto a remote node. As usual, the parent is returned the PID of its new child, but note that this will reflect the number of the node on which the child has been forked.

In all respects, the parent and child behave as if they are co-located: the child inherits its parent's process group id (PGID), session id (SID), text and data segments, open files etc. The parent can wait on its remote child in the usual way.

An additional form of this call, `rforkmulti()`, is provided to fork multiple children at once on a specified set of remote nodes as a more efficient alternative to a series of `rfork()`s.

`rexecve()`

Taking an additional node parameter over the conventional `execve()` system call, `rexecve()` executes a new program image on a remote node. Note that the PID is retained when the process moves to the new node – it is, of course, still the same process.

`migrate()`

This system call relocates the calling process to the remote node specified by a single parameter. Upon successful return, the caller will be executing on the new node but in all other respects the process is unaltered: it has the same PID, PGID and SID, text and data, open files etc.

`kill3()`

This system call is a superset of the usual `kill()` system call. Taking an additional argument, `kill3()` sends a signal and the associated integer argument to a specified process. This semantic is of relevance only to the TNC-specific signal `SIGMIGRATE`: the default handling of which is to migrate the receiving process to the node indicated by the associated argument. Hence, `kill3()` may be used to migrate any process regardless of whether it is TNC-aware; the load-leveling daemon employs this mechanism to migrate processes away from loaded nodes.

`node_self()`

This system call returns the number of the node where the caller is executing.

TNC User Commands

TNC provides only three additional user commands which the Zoo suite exploits; these are:

onnode

This command executes a command on a specific node. `rexecve()` is used to execute the given command. Standard input, output and error are inherited from the invoking shell in the usual manner. For example,

```
onnode 3 ls -l
```

will perform an "ls -l" command on node 3.

kill3

This command acts like **kill** but takes an additional argument which specifies the extra parameter required by the `SIGMIGRATE` signal to indicate a node number to which a designated process is to migrate. For example,

```
kill3 -SIGMIGRATE 1 $$
```

will migrate the invoking shell to node number 1.

node_self

This command prints to standard output the node number on which the command is executed. For example,

```
onnode 10 node_self
```

prints 10.

Evolution

The members of Zoo evolved from a pair of TNC test programs called **frog** and **bunny**. **frog** exercised TNC's `migrate()` system call by successively hopping to each node in a list as fast as it could. **bunny** did much the same but was a little more amusing because it paused on each node to "eat" cpu time (or as an option, sleep) for a random length of time. Note that the evolution of randomness was a very important attribute indeed.

Apparently worried by the infestation of small animal life at this stage, the Locus VP responsible for the project suggested that a predator should be introduced to maintain the ecological balance. He suggested that a hawk was necessary to control those darn bunnies. You can see why some get to VP level while others simply stay home and feed the animals.

Two strains of **frog** and **bunny** now developed – one for common or garden AT386s and the other for the more exotic land of the i860 Hypercube. Moreover, each strain needed to be raised separately. Around the same time, the project architect (a keen Korn shell fancier) had ported a version of the Korn shell to the Hypercube. So it was decided that it the Zoo should specialize in members of the portable ksh script genus.

Being rather domesticated, ksh scripts are adaptable, are easily maintained and can be highly intelligent – quite apart from being good travelers. Korn has built-in randomness, too. Naturally, scripts aren't particularly fast and together with their shell they're large – but for the purposes of exerting stress, these are indeed quite desirable traits.

Hence, **frog** and **bunny** were succeeded by **bee** which buzzes from node to node working at each stop, but does so in the form of a script using the TNC commands. Of course, the name **bee** doesn't correctly reflect the size of the beast – so think *bumble*-bee.

Even in the form of a shell script, individual subtests like **bee** impose very little real stress on a system. However, with an increasing population, increased loads and random movements, stress is built. A swarm of bees proved to be quite a challenging test for TNC.

The coordination of a population of subtests is vested in a master, god-like, script called **pan**. **pan**, represents the controlling influence over a test: it shepherds the life and death of a set of replicated child subtests. Different levels of stress may be produced, in specific places and over specific time periods by varying the options given to **pan**.

pan represents the central idea in the Zoo suite: you might say it's the Zoo's theosophy.

The following section describes in more detail the functions of **pan** and other test scripts.

Description of the Species

This section describes the behavior of the main members of the Zoo test menagerie. Note that common options are fully described once on their first occurrence and are only listed thereafter. Furthermore, options are subject to the general convention that uppercase flags request fixed behavior while lowercase signifies randomness. For example, "-T 30" specifies a fixed time period of 30 seconds, whereas "-t 30" specifies a random time up to a maximum of 30 seconds.

`frog <node_list>`

frog leaps through the nodes given by <node_list>, a space-separated list of node numbers. Additionally, **frog** may be given a repeat count, specified by the -n option.

frog is a simple C program originally intended to exercise the `migrate()` system call.

Options:

-n <count> leap through nodes <count> times

`bunny [options] <node_list>`

bunny hops through the nodes given by <node_list>, a space-separated list of node numbers. On each node, **bunny** pauses either to sleep or eat (cpu) for either a fixed time or, alternatively, a random length of time. It may be requested to repeat its wanderings for a specified number of times.

bunny is a C program. It's purpose is to test interactions between the `migrate()` system call and signal catching in the user task, and to verify that outstanding timers are migrated correctly.

Options [-e|E|s|S -n -N -r -h -v|V] with:

-E <num> eat cpu for a <num> seconds on each node

-e <num> eat cpu for a random time < <num> secs

-S <num> sleep for <num> seconds on each node

-s <num> sleep for a random time < <num> secs

-n <count> hop through nodes <count> times

-N <len> generate a random list of length <len> with node numbers in the range 0..<node1>

-r <num> use <num> as the seed for randomness

-h issue `migrate()` calls from timer signal catcher

-v catch `SIGMIGRATE` but don't ignore

-V catch and ignore `SIGMIGRATE`

`bee [options] <node_list>`

bee buzzes among a list of nodes either until all nodes have been visited or, optionally, for a specified period of time. <node_list> may contain ranges of nodes in the form "node1..node2". Nodes are visited by default in the order specified by the list which is used cyclically if a time period is

given; optionally, nodes are chosen in random order. On each node **bee** pauses to work (-e or -E) or sleep (-s or -S) for a given time (either a fixed time or one randomly chosen up to a maximum).

bee is a Korn shell script which tests the distributed relationship and signaling handing of TNC. In particular, subshells demonstrate parent child relationships. The **kill3** command is used sting itself with a `SIGMIGRATE` signal in order to migrate between nodes.

Options [-e|E|s|S -L -n|N -q -r -t|T] with:

-L <file> log to file instead of standard output.

-N nodes to be chosen from list in the order specified

-n nodes to be chosen from list randomly

-q disable logging messages

-T <num> number of seconds to live

-t <num> randomly determined lifetime up to this limit

Example:

```
bee -T60 -e10 0..3
```

Buzzes for 1 minute between nodes 0, 1, 2 and 3 eating for a random time up to a maximum of 10 seconds at each stop.

worm [options] <node_list>

worm squirms about a list of nodes for a specified period of time as it munches through the pages of a file – it's a *book*worm.

worm is another script written to test that simple file operations, and particularly file offsets, are maintained during migrations induced asynchronously by delivery of SIGMIGRATE.

Firstly, a file of specified length is created before **worm** wanders from node to node reading it. As each line is read, **worm** checks that it has read what it expects. On reaching the end, it starts over again.

Options [-e|E -f -L -l -n|N -p -q -r -t|T] with:

-f <file> filename to write (defaults to /tmp/worm.$$)

-l <length> specifies the number of lines to be written in the file

-p <text> text with which lines are to be padded

Example:

```
worm -t60 -e10 -l1000 0 1 2 3
```

Munches for up to 1 minute between nodes 0..3 eating through a file of 1000 lines with a random stay of up to a maximum of 10 seconds on each node.

duck [options] <node_list>

duck migrates between nodes performing floating operations. At each stop, duck feeds on pi: that is, pi is calculated. As you might expect from a bird brain, the calculation is not optimal; in fact, it's very random. **duck** generates a series of random points (x,y) in a unit square and calculates the square of the distance of each point from the origin (x*x+y*y). The ratio of the number of coordinates within unit distance of the origin to the total number of coordinates generated turns out to be, by the law of averages, approximately pi/4.

duck is a combination of C-coded program to do the numbers and a script to manage the migration. Its purpose is to verify that floating point operations are preserved during migration. To ascertain this, duck calculates pi first (to a certain number of terms) without migrating, then repeats the same calculation while migrating between nodes, and finally compares the two results.

duck is currently valid only for the HyperCube because floating point unit state preservation is not implemented on 386 platforms.

Options [-e|E -h -i -L -n|N -q -r] with:

-i <num> number of iterations to perform

-h calculate pi using the sum of the harmonic series (1 - 1/3 + 1/5 - 1/7 ...)

Example:

```
duck -i1000000 -e5 1..7
```

Calculates pi by considering one million random points while migrating between nodes 1, 2, 3, 4, 5, 6 and 7, calculating for a random time of up to 5 seconds at each stop.

artemis [options] <node_list>

artemis, named after the Greek goddess of hunting, is a generic predator which stalks prey amongst a list of nodes for a specified period of time. On each node, **artemis** searches for commands matching its list of prey and these it attempts to kill. If a kill is successful, **artemis** pauses to eat for a random (-d) or fixed time (-D) before moving on. If there is no prey or the kill attempt fails, **artemis** sleeps for a random (-d) or fixed time (-D).

artemis is a Korn shell script which exercises TNC's distributed signal handling. The **kill3** command is used to send itself a `SIGMIGRATE` signal to cause migration between nodes; naturally, **kill** is used to kill prey.

Options [-d|D -L -n|N -p -q -r -t|T] with:

-D <num> number seconds to delay between migrations

-d <num> maximum number seconds to delay a random time between migrations

-p <prey> space-separated list of the names of prey to kill

Examples:

```
hawk() {
      artemis -p"bunny" $*
}
hawk -t240 -d10 -n 3 4 5
```

Defines "hawk" to be a bunny-eating predator which is released to hunt for 4 minutes on nodes 3..5, pausing for up to 10 seconds at each place before flying elsewhere.

```
crow() {
        artemis -p"frog bee worm" $*
}
crow -t120 -D10 1
```

Defines "crow" to be a frog, bee and worm-eating predator which is to hunt for 2 minutes exclusively on node 1, pausing exactly 10 seconds between each meal.

pan <options> cmd

pan[1], named after the Greek god of flocks and herds, spawns (using the **onnode** command) a specified number copies of a command cmd in parallel over a given list of nodes. Each test is run a given number of times in series. Hence, **pan** is responsible for maintaining a constant population of the given test. The node on which each test is remotely executed is chosen either deterministically (-N) or randomly (-n). To illustrate this, the command:

```
pan -N"0 1 2" -p3 -g4 who
```

says: run 3 clones of the who command on the 3 nodes 0, 1 and 2 and repeat each 4 times. To picture what's happening, think of this as a forming a matrix over time:

node \ time	1	2	3	4
0	who	who	who	who
1	who	who	who	who
2	who	who	who	who

This is the simple case, but one that is useful for testing the load leveling since it continually generates a workload on a subset of nodes and you would expect the leveler to migrate some of it to other nodes.

If, instead of the first example, the list of nodes were to be given with the (lower-case) -n option, each instance of the command will be `rexec()`ed on a random node from the list. So, what you might see now is:

node \ time	1	2	3	4
0	who	who who		
1	who		who who who	who
2	who	who		who who

Much more like what you'd expect from a family of caged owls.

Options [-L -g -n|N -p -q -r] with:

-g <num> number of generations, i.e. how many times to repeat each clone

-N <list> space-separated list of node numbers to be used cyclically

-n <list> space-separated list of node numbers to be used randomly

-p <num> the size of the population, i.e. parallel copies to spawn

Examples:

```
pan -p3 -g2 -N0..2 date
```

This spawns 3 copies of the date command in parallel on nodes 0, 1 and 2, repeating each twice. Useful only to white rabbits, perhaps: "Oh, my goodness, I'm late, I'm late, I'm late!".

```
pan -p5 -g100 -n"1 3" \
    bee -T30 -e10 1 3
```

This unleashes a swarm of a total of 500 bees, 5 at a time with each **bee** buzzing for 30 seconds randomly between nodes 1 and 3.

```
pan -p5 -g100 -n"1 3" \
    bee -T30 -e10 1 3 &
pan -p2 -g10 -n"1 3" \
    artemis -T300 -p"bee" 1 3
```

Here the swarm of bees is kept in check by two bee-eating predators hunting in the same territory.

```
pan -g10 -p2 -N"1 3" \
    pan -p5 -g100 -n1 \
        bee -T30 -e10 1 3
```

Finally, **pan** demonstrates true supernatural powers by invoking himself to create ten generations of two distinct swarms.

Logging

All zoo members leave entrails which can be usefully examined should untimely death occur. Messages are sent to standard output by default but can be redirected to a file using the -L option or disabled altogether with the quiet option, -q. Messages are in a standard form:

[1] When Pan appeared, mortals took fright - hence panic.

`<time> [<node>.<pid>] <name>: <text>`

where:

- <time> is a timestamp of the form `hh:mm:ss`
- <node> is the node number from which the message was issued
- <pid> is the process id of the process issuing the message
- <name> is the name of the process
- <text> is the text of the message

Most scripts generate messages when a significant change occurs. For example, **bunny** reports when each hop is attempted and whether its about to eat or sleep and for how long; see Figure 1.

Observations and Results

The results of developing and using the Zoo suite fall into three categories: what was learned by developing and running individual tests; what further was learned as stress was developed, and what deficiencies were apparent in the tests themselves.

The majority of the following results were obtained on the i386 platform since OSF/1 AD TNC on the HyperCube has until only recently been too unstable to support stress tests.

Individual tests

The individual tests (**frog, bunny, bee, worm** and **duck**) sought to test specific aspects of TNC but to do so in an increasingly flexible way. Hence, the transition from programs coded in C to Korn shell scripts.

frog, testing repeated calls to the `migrate()` system call, turned out to be unsuccessful (if that's the right way to look at it) in exposing any problem in server code. However, it has consistently revealed problems in the NORMA multicomputer support of the Mach microkernel, which was being developed concurrently with TNC. Specifically, the management of ports which were migrating rapidly between nodes proved to be a non-trivial problem.

bunny had more success in digging holes in the TNC server code, especially on the HyperCube. **bunny** had considerable fun with the signal delivery trampoline code associated with SIGMIGRATE. And even until very recently, **bunny** proved able to confuse and exhaust Mach. However, the major advance that **bunny** brought was the introduction of random behavior: randomness proved to be a crucial characteristic of the Zoo suite.

Progressing from **bunny**, **bee** was the first TNC test developed as a Korn shell script. It set out simply to emulate the general behavior of **bunny**, without implementing some of **bunny**'s esoteric behavioral traits, and it was clearly able to do this. However, the adaptability as a script proved to be very powerful. With only two specialized user commands (**kill3** and **node_self**), a test of distributed processing could be created.

worm was devised to check that file offsets were maintained correctly during process migration, and they were. The test inevitably became less interesting and has been used infrequently since. Nevertheless, worm demonstrated once again what can be achieved as a script.

duck, the most recent addition to the Zoo suite, revealed a problem in the Hypercube implementation failing to fully account for the pipelined floating point architecture of the i860 processor.

artemis proved to be a poor hunter, indeed. As a script and relying on the standard **ps** command to spot her prey, **artemis** was, quite simply, too slow and shortsighted. By the time she had recognized a prey and made an attempt at a kill, the prey was long gone. Even a lucky shot might only "wing" a prey by killing a subshell but leaving the body to escape. To become more mortal (i.e. deadly), **artemis** would need to be provided with less mortal (i.e. more godlike) powers unavailable to a script. **artemis** contributed only fun to the suite.

Stress and stress-induced problems

Randomly behaved tests implemented as Korn shell scripts were two key elements of the Zoo suite. But to generate real stress, a third element was required: running combinations of tests distributed over multicomputer nodes; **pan** provided this.

The invocation of pan that has proved particularly stressful is referred to as the swarm of killer bees. Slight variations of this have been repeatedly (and infuriatingly) successful in provoking server panic or deadlock. In fact, so effective has the swarm proved as a stress test that little time has

```
# bunny -e10 -n2 1 0
17:41:07 [0.72] bunny: bunny hopping to nodes 1 0, 2 times
                       eating for a random time < 10 seconds on each node
17:41:07 [0.72] bunny: eating on node 0 for 1 secs...
17:45:23 [1.72] bunny: eating on node 1 for 4 secs...
17:41:16 [0.72] bunny: eating on node 0 for 2 secs...
17:45:33 [1.72] bunny: eating on node 1 for 9 secs...
17:41:31 [0.72] bunny: done on node 0
```

Figure 1: Significant change event log

been given to other combinations of tests. An example of test log for a failing run would be as shown in Figure 2.

A point of interest to note here is that OSF/1 AD TNC was debugged on AT386-class machines with the assistance of a fully symbolic debugger, **gdb**. Since the OSF1/ AD TNC server is merely a user task under Mach, it can be run as as a second Unix server running alongside another (potentially itself but in fact BSD4.3 in our case). It is loaded as a unix process under the first server whence it can be symbolically debugged as a user process (albeit multithreaded).

A review of some problem areas exposed by swarm test runs is depressing but instructive:

MP deadlock

Many instances of deadlock were recorded due to failure to observe the multiprocessing lock hierarchy. Note that since the OSF/1 AD TNC server runs as a Mach user task it is subject to pre-emptive thread scheduling and so MP locking is required even where there is a single CPU present. TNC added a virtual process (VPROC) locking hierarchy above that existing in the base OSF/1 AD server. In particular, OSF/1 AD employs a master lock which cannot be taken until **all** VPROC locks are acquired. However, a handful of code paths intersecting the base server and VPROC layer turned up failing to comply with this rule.

Emulator/server interaction

A blemish in the architecture of OSF/1 AD is the existence of the "emulator". This a raft of code lives in user space and acts as agent for the server. The emulator fields system call traps from the user program and performs any necessary RPCs to process management and/or file servers to honor the calls. In the configuration used in TNC, memory is shared between emulator and process management server to minimize necessity for RPCs. Certain shared read/write access to this area proved to be deficient under stress.

Fileserver signal processing

When a signal is to be delivered to a process that is performing a file operation, it is forwarded from the process management server via the emulator to the fileserver involved. Even on a single node system process management, emulator and fileserver threads are distinct and, under stress, the fileserver was found to be mishandling race conditions between forwarded kill signals and process destruction (communicated by means of no-more-sender notification for the file port being closed).

Note: The problems above were either inherent in the underlying server, or a result of TNC's interaction with it, and not a product of distributed processing. Indeed, they were apparent on a single node system. However, the following problems implicated the remote processing within TNC.

```
pan -h3 -g10 -N"0 1" bee -t30 -e10 0 1
11:02:28 [0.61] pan: [73] 2-1/10 - onnode 1 bee -t30 -e10 0 1 &
11:02:28 [0.61] pan: [69] 3-1/10 - onnode 0 bee -t30 -e10 0 1 &
11:02:28 [0.61] pan: waiting for 3 clones of "bee -t30 -e10 0 1" to complete
11:02:28 [0.61] pan: [70] 1-1/10 - onnode 0 bee -t30 -e10 0 1 &
11:02:31 [0.69] bee: buzzing around for 12 seconds...
11:02:31 [0.70] bee: buzzing around for 27 seconds...
11:02:32 [0.70] bee: buzzing to node 0
11:02:34 [0.70] bee: eating on node 0 for 4 secs
[...lines deleted...]
11:03:06 [0.61] pan: [137] 1-2/10 - onnode 0 bee -t30 -e10 0 1 &
11:03:07 [0.137] bee: buzzing around for 8 seconds...
11:03:07 [0.137] bee: buzzing to node 0
11:07:25 [1.109] bee: eating on node 1 for 1 secs
11:03:08 [0.137] bee: eating on node 0 for 9 secs
11:07:27 [1.73] bee: ...expiring on node 1
11:03:11 [0.61] pan: [152] 2-2/10 - onnode 1 bee -t30 -e10 0 1 &
11:07:30 [1.109] bee: buzzing to node 0
[node: 0] panic: norma_get_nameserver_port failed 0x4
syncing disks... 4 4 4 3 3 3 2 2 2 2 1 1 1 1 done
Debugger (suspending server pid=276)
```

Figure 2: Test log for a failing run

Unreaped remote orphan processes

At one stage of development, after a round of stress tests had been run, nodes other than where **init** resided were cluttered by zombie children bequeathed to init by exiting parent processes. This was diagnosed to be a race between parent and child termination wherein **init** was mistakenly lead to believe that it had adopted running (remote) processes.

Migrated process ignoring signals

A migrated process would occasionally ignore signals. Perversely, this resulted from too much process state information being transferred to the new node. If a process with its program counter in user space had a signal outstanding at migration time, the old server would have dedicated a separate thread to handle the signal. However, migration would abort this thread and no equivalent thread would be required on the new node. However, a record of the thread's existence was transferred and consequently the new server was left to believe that a separate thread would deal with signals – but this not being the case, all subsequent signals were ignored.

Mach message sequence counting

In a multithreaded Mach environment with TNC transferring Mach ports between nodes, race conditions are possible between one server thread receiving a message on one port and another thread moving or destroying that port. An important consideration here is that reception of a Mach message is not an atomic operation (microkernel threads can preempt each other).

If no other steps are taken, it is possible to receive a message addressed to a null port (`MACH_PORT_NULL`) if the port is destroyed after a message is dequeued by the microkernel but before it is returned into user space. Mach provides a solution to this problem by assigning sequence numbers to received messages. Hence it can be established if there are receives in progress so that a port is not moved or destroyed until it is safe.

The OSF/1 AD TNC server equates Mach ports to data structures and consequently receiving a message addressed to a null port is panic-worthy. TNC failed to exploit sequence counts; this omission was exposed when race conditions emerged under stress.

Limitations of using shell scripts

A problem apparently connected with TNC delivering signals to a process undergoing migration cropped up under stress and remained outstanding for a long time. It was eventually tracked down to a failure of the Korn shell to process a user-defined trap reliably. As first implemented, scripts used the **trap** command to catch `SIGUSR2` signals conveying timeouts. Very occasionally, such a signal was not caught if sent when the target shell was migrating. However, after exhaustive tracing through signal handling code, it was proved that no signal loss was occurring in TNC and that the signal was always delivered into the shell's signal catcher only to be dropped thereafter.

After modifying the implementation of timeouts, this problem was eliminated. Clearly, a shell cannot be expected to perform asynchronous (and certainly not multithreaded) functions of which a specially written program is capable.

Beware that, although the Korn shell provides a rich programming environment, some features are absent from earlier versions. Consequently, the goal of portability must limit scripts to use only the features of **ksh-i** or the 6/3/86 version of the Korn shell; refer further to "The Kornshell", Bolsky/Korn 1989.

Random reproducibility

A general observation to make about stress testing with random components is not to expect reproducible results. This is especially true when several latent bugs exist. During Zoo testing, it was typical for one problem to be seen but for a completely different symptom or bug to occur when the same test was repeated.

Stress-induced problems typically depend on timing factors that will be random. All that can be relied upon is that the more serious problems recur with a higher probability; the worse problem will dominate. There's a Darwinian phenomenon at play here: the weakest (worst) bug will tend to be killed off first. Nevertheless, it is often possible to weight probabilities in favor of one bug occurring over another by varying the parameters given to a stress test.

Only the most severe problems reproduce reliably. For more rare problems, it was important adopt a flexible approach to debugging. Do not attempt to go after one bug at a time but instead pursue each problem as it occurs. If no conclusion can be drawn immediately, and a test requires repeating, don't assume that the same problem will recur.

Conclusions

The Zoo tests have been particularly successful in stressing the multithreaded and multiprocessing aspects of OSF/1 AD TNC and the supporting microkernel. The MP and distributed locking algorithms implemented in TNC have been confidently validated. Many subtle race conditions and combinatorial locking problems have been revealed and corrected over the course of very few months of effort.

The approach to stress testing described in this paper provides a flexible framework in which tests can be written on an individual basis to verify a specific functional area and then amplified by random distribution and replication to form a true stress test.

The Zoo suite demonstrates that a complex and powerful multicomputer architecture can be stressed by a familiar shell environment and just a few hundred lines of script supplemented a very small number of specialized but simple user-level commands.

I hope also that Zoo demonstrates that advanced computing can and should be fun. Unix has developed in a spirit of adventure and expression, and long may it continue.

Acknowledgements

I thank Phil Shevrin, the unnamed Locus VP, for suggesting the paper; Brad Kemp for coining the name "Zoo"; and Roman Zajcew for using ksh. I also thank David Black, Steve Sears and Alan Langerman at OSF for making Mach a much happier place for bunnies (and all other lifeforms).

Author Information

Chris Peak once received a degree in Mathematics from Cambridge University. He is currently a Consulting Member of Technical Staff with Locus Computing Corporation in San Diego. Before moving to the US, he worked for British systems house Logica. He is married and has a 2-year old son who likes bunnies. He is a member of the Zoological Society of San Diego. He may be reached electronetically at chrisp@locus.com.

Appendix: Sample Script Listings

The Korn shell script for **bee** and a shell library file, stresslib, are included here for reference.

bee

```
#!/bin/ksh
#
# HISTORY
# $Log: bee,v $
# Revision 3.2  92/11/18  11:39:36  chrisp
# Major overhaul: now use stresslib, accept node ranges, various other tweeks.
#
# Revision 3.1  92/09/21  14:18:34  chrisp
# Change the -t n option to generate a random lifetime in the range 1..n secs.
# The -T n option should now be used to specify a fixed lifetime of n secs.
#
# Revision 3.0  92/07/21  12:06:11  chrisp
# First appearance
#
#
. stresslib

EAT_OR_SLEEP=eat

while getopts ":L:E:e:Nnr:S:s:T:t:q" opt; do
    case "$opt" in
        L)  LOGFILE="$OPTARG";;
        E)  EAT_OR_SLEEP=eat; DELAY=$OPTARG;;
        e)  EAT_OR_SLEEP=eat; DELAY=$OPTARG; randomizing_delay=true;;
        N)  RANDOMIZING_NODES=;;
        n)  RANDOMIZING_NODES=true;;
        q)  QUIET=please;;
        r)  RANDOM=$OPTARG;;
        S)  EAT_OR_SLEEP=sleep; DELAY=$OPTARG;;
        s)  EAT_OR_SLEEP=sleep; DELAY=$OPTARG; randomizing_delay=true;;
        T)  TIME=$OPTARG;;
        t)  let "TIME=(RANDOM % OPTARG) + 1";;
        :)  echo "$NAME: $OPTARG requires a value"
            exit 2;;
       \?)  echo "$NAME: unknown option $OPTARG"
            echo "usage: $NAME -n -t|T<secs> -e|E|s|S<secs> <node_list>"
             exit 2;;
    esac
done
```

```
shift OPTIND-1
node_list="$*"

#
# Prompt for all the important stuff if not given
#
[ "$node_list" ] || read node_list?"Node list? "
[ "$node_list" ] || node_list="$(node_self)"

declare_nodes $node_list

trap 'stop_eating; stop_timing; exit' INT KILL TERM
if [ "$TIME" ]; then
    CYCLING_NODES=true
    start_timing $TIME
    log "buzzing around nodes ${NODES[*]} for $TIME seconds..."
else
    CYCLING_NODES=
    log "buzzing through nodes ${NODES[*]}..."
fi

#
# Take a random walk through the nodes,
#  pausing to do a random amount of work on each.
#
let "delay_time = DELAY"
while timing; do
    #
    # Make a (random) selection from the list of nodes
    #    and migrate there
    #
    next_node
    [ "$NODE" ] || break
    log "buzzing to node $NODE"
    migrate_to_node $NODE

    [ $DELAY -eq 0 ] && continue

    #
    # Eat cpu for a time
    #
    if [ "$randomizing_delay" ]; then
        let "delay_time = (RANDOM % DELAY) + 1"
    fi
    log "${EAT_OR_SLEEP}ing on node $(node_self) for $delay_time secs"
    $EAT_OR_SLEEP $delay_time
done

log "...expiring on node $(node_self)"
exit 0
```

stresslib

```
#!/bin/ksh
#
# HISTORY
# $Log: stresslib,v $
# Revision 3.0  92/11/18  11:32:23  chrisp
# Created as library for functions common to the Zoo stress suite.
#
#

RANDOM=$$            # default random seed
LOGFILE=/dev/tty     # log to stdout by default
```

```
NAME=${0##*/}

#
# Function to check that required commands are in path.
#
required() {
    for cmd in $*; do
         [ "$(whence $cmd)" ] ||
             { echo "$NAME: can't fin $cmd"; exit 2; }
    done
}

#
# Function to create node array from list of nodes
#
declare_nodes() {
    let "num_nodes = 0"
    for node in $*; do
        case $node in
             *[!0-9.]*)
                log "bad node number specified"
                exit 2
                ;;
             *..*)
                lo_range=${node%%..*}
                hi_range=${node##*..}
                while ((lo_range <= hi_range)); do
                    let "NODES[num_nodes]=lo_range"
                    let "lo_range += 1"
                    let "num_nodes += 1"
                    done
                ;;
             [!0-9])
                log "bad node number specified"
                exit 2
                ;;
             *)
                let "NODES[num_nodes]=node"
                let "num_nodes += 1"
                ;;
        esac
    done
    let "node_index = -1"
}
#
# Function to select the next node from a list
#  either randomly or cyclically
#
next_node() {
    if [ "$RANDOMIZING_NODES" ]; then
         let "node_index = RANDOM % num_nodes"
    else
         let "node_index += 1"
         if [ "$CYCLING_NODES" ]; then
            let "node_index = node_index % num_nodes"
         fi
    fi
    NODE=${NODES[node_index]}
}
```

```
#
# Function to migrate a process (default ourself) to a new node
#
migrate_to_node() {
    node_to=$1
    process=${2-$$}
    node_from=$(node_self)
    if [ $node_to != $node_from ]; then
        kill3 -SIGMIGRATE $node_to $process 1>/dev/null 2>&1
    fi
}
#
# Function to put out progress announcements:
#
log() {
    text="$1"
    set 'date'
    time=$4
    [ "$QUIET" ] || echo "$time [$(node_self).$$] $NAME: $text" >>$LOGFILE
}
#
# Functions to start, stop and test for outstanding timer
#
start_timing() {
    sleep $1 &
    TIMER=$!
}
stop_timing() {
    kill $TIMER 1>/dev/null 2>&1
}
timing() {
    if [ "$TIMER" ]; then
        kill -0 $TIMER 1>/dev/null 2>&1
        return $?
    else
        return 0
    fi
}
#
# Function to eat cpu for a given time
#
eat() {
    ( while : ; do
        :
      done
    ) &
    EATER=$!
    sleep $1
    kill $EATER
}
stop_eating() {
    kill $EATER 1>/dev/null 2>&1
}
required kill3 node_self
```

PhoneStation, Moving the Telephone onto the Virtual Desktop

Stephen A. Uhler – Bellcore

ABSTRACT

PhoneStation is a system that provides a Sun Microsystems SPARCstation with complete control over an ordinary telephone line. It consists of a telephone line interface unit with loop control and touch tone detection, a suite of supporting software libraries that include digital signal processing for call progress monitoring, text-to-speech conversion, telephone line control, and PhoneScript, a high level procedural language that uses TCL for building interactive telephone based applications.

Introduction

For over a decade now, the workstation has been viewed as an electronic desktop, with multiple windows on the computer screen as the metaphor for a desk [1]. This electronic desktop has become the focus for dealing with office information. The telephone, although an important component of an actual desktop, has not yet been integrated into the modern desktop environment.

It should be possible to receive audio telephone messages as ordinary electronic mail (email), thereby taking advantage of the many message management capabilities we have become accustomed to in email systems. A unified interface to handle voice mail and email would eliminate the distinct and increasingly more complex user interfaces to telephone answering machines or voice mail systems, and provide the ready exchange of information between the computer and the telephone.

With the telephone an integral part of the computer desktop, many new applications come to mind. While retrieving voice mail messages over the telephone why not have the answering machine application convert your regular email to speech, and read it to you as well. If there is a fax machine available, as is the usual case at a hotel or conference, you could instruct the answering machine application to have the workstation fax you your regular email. Once on the phone, connected to your workstation, why not fax that article you forgot to bring with you, or that viewgraph you didn't think you'd need.

This paper will describe the components of PhoneStation, a system that provides a Sun SPARCstation with complete control over an ordinary telephone line. After briefly describing the PhoneStation hardware and basic software facilities, it will describe in detail, PhoneScript, the PhoneStation high level language for building interactive telephone applications.

PhoneStation System Components

PhoneStation runs on a Sun Microsystems SPARCstation. The system consists of some hardware "glue" that enables the SPARCstation to interface to a telephone line, a suite of software support libraries, and PhoneScript, a language for building interactive telephone applications. The software components of PhoneStation are shown as boxes in Figure 1. The basic support libraries are along the bottom. Applications programs are normally written in the PhoneScript language, although they can be written in C, and call the underlying library routines directly.

PhoneStation Hardware

The SPARCstation hardware interface, called STIM (SPARCstation Telephone Interface Module), connects to the SPARCstation through a serial port and the audio connector. It is assembled from off-the-shelf components, and fits in a 2" x 4" x 4" box. A block diagram of the STIM hardware is shown in Figure 2.

The core of the STIM is the single chip computer, a Zilog Z8 [2]. The Z8 has 16 individually controllable I/O (input/output) lines, three of which are configured as an rs232 serial port. The remaining I/O lines are connected to a telephone line interface hybrid, a touch-tone detection and generation chip, a telephone loop current detection relay, and a pair of audio switching relays. The telephone line interface unit provides the required isolation from the telephone line. In addition to inserting and extracting audio signals from the telephone line, it detects ringing, and can place the telephone line in either the on-hook or off-hook state. The touch-tone detection chip does just that; detect the presents of touch-tones, which are converted to ASCII signals by the Z8, and sent over the serial interface to the SPARCstation. The loop sense relay monitors the state of the telephone line to detect when a telephone call has terminated. The audio switching relays permit other audio devices to be connected to the SPARCstation when the STIM is not in use.

The program that runs in the Z8, written in *basic,* communicates with a process on the SPARCstation using single letter ASCII commands via the serial port. The digital-to-analog and analog-to-digital conversion capabilities of the SPARCstation are used to play and record digitized audio.

PhoneStation Software

The primary application interface to PhoneStation is PhoneScript, a command interpreter that uses TCL (Tool Command Language) [3]. TCL, written by John Ousterhout of Berkeley, is a freely available library of C routines that provide a software system with an embeddable shell-like command interpreter. This command interpreter is combined with a suite of software support libraries, written in C, providing digital signal processing (DSP) for call progress monitoring, a text to speech synthesizer using the ORATOR® speech synthesizer [4], and some relational file management routines that facilitate the storage and retrieval of data that may be required by a telephone application.

PhoneStation consists of 5000 lines of C code in the support libraries, and another 3000 lines of C code to interface them with TCL. There is another 250 lines of Basic that runs on the Z8 microprocessor in the STIM, as well as 1500 lines of C code that provides the development environment for programming the Z8 and configuring the DSP code.

The telephone interface module provides a device independent abstraction for interacting with the telephone line. It interacts with the STIM over a serial line. The software configures the SPARCstation serial line same way as a modem that is set up to permit both incoming and outgoing calls. An application that is waiting for incoming telephone call blocks (in open()) until a call arrives. When the STIM detects ringing on the telephone line, the Data Carrier Detect line of the serial interface is asserted, causing the open() to complete, and the application to continue. Additional information about the state of the telephone line is then passed between the STIM and the telephone interface module over the serial interface. Applications wishing to place

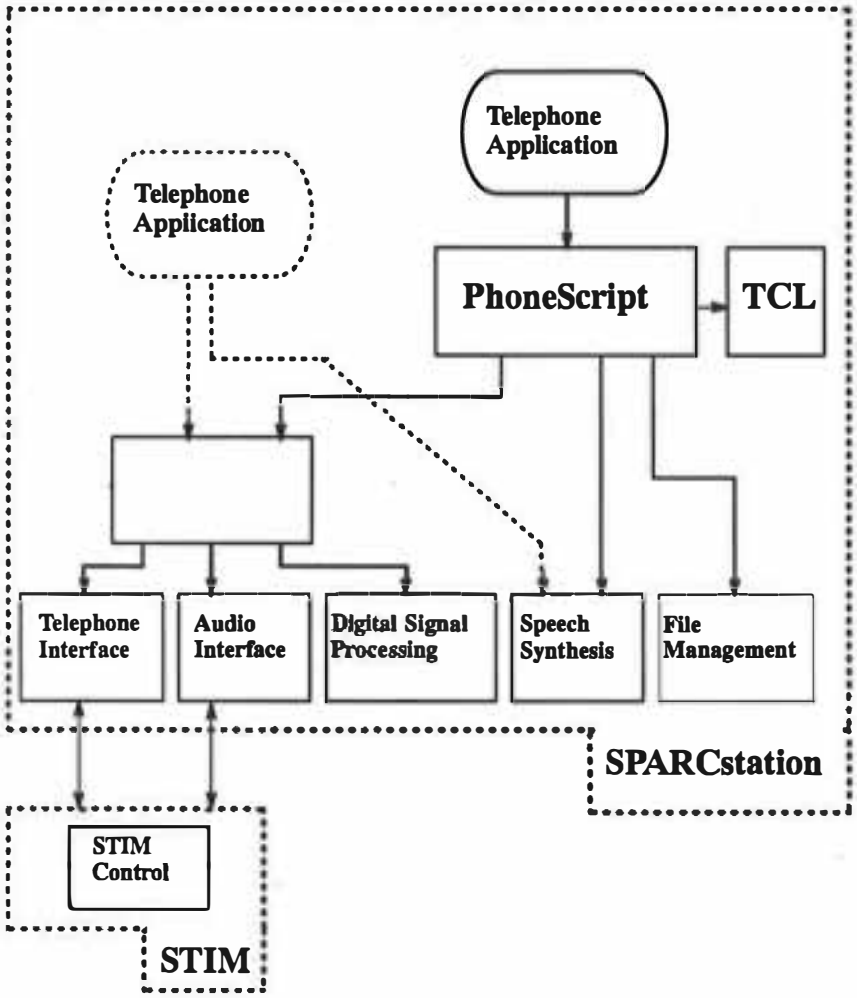


Figure 1: PhoneStation Software Components

outgoing calls can do so any time the telephone interface is not already being used, even if another application is waiting for an incoming call.

The audio interface module controls access to the SPARCstation audio device. It set the play and record volume levels, and controls the amount of audio data in the audio device driver queues. PhoneStation plays audio files by periodically sending batches of audio to the device queue. Application programs can change the batching interval to obtain more time for other computations before the next batch of audio is required.

The DSP module uses second order recursive digital band pass filters and energy detectors [5], running in software on the SPARCstation, to process the incoming audio stream and determine the status of a telephone call. Signaling tones used for telephony are simple combinations of pure tones (sine waves). The band pass filters isolate the sine waves, then the energy detectors determine if a signal is present at the required frequency. The DSP module identifies dial tone, ringing, and busy signals, which are used to monitor the progress of an outgoing call. Modem tones and voice patterns are recognized once the call has been completed. Routines are available to detect and decode touch-tones as well, even though in the current version of PhoneStation, the touch-tone detection can also be done by the STIM in hardware. The signaling tones, as well as various answering-machine like "beeps," are synthesized by the digital signal processing module as needed.

The text-to-speech synthesizer runs as a background process, and has been optimized to pronounce names and addresses accurately, although it can synthesize arbitrary text quite well with a little coaching. The synthesizer typically takes less time to synthesize an utterance than it takes to speak it. Synthesized output can either be sent to the telephone directly, or saved in a file for later use.

The file management module provides a simple relational abstraction of a file that integrates structured file access into the PhoneScript language. It provides access to the files contents through TCL variables, and supports the selection of items in the files through the evaluation of TCL expressions containing references to specific items.

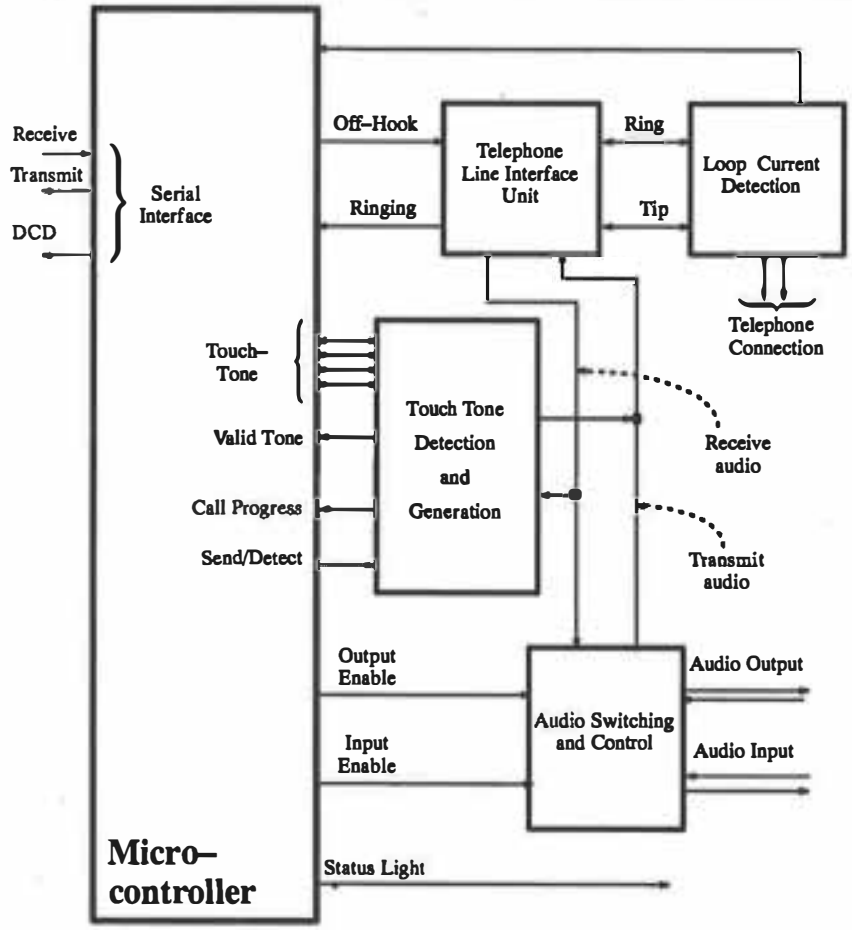


Figure 2: STIM Hardware

The PhoneScript Language

Telephone applications are similar to many real time process control applications. They have real time constraints; the phone must be "answered" within a certain time, or a touch-tone received from the user must be processed before the next one arrives. Time out conditions abound: how many rings to wait before "answering" the telephone, how long to wait for a dial tone, and how much time to listen for a touch-tone, are examples of just a few. Most of the inputs into the system come in the form of asynchronous events, they can occur at any time, and often do.

A typical method for dealing with this type of system in a language such as C, is to use an event driven state machine. The program waits for an event, acts upon it, transitions to the next state, and waits for the next event to occur. Although state machines can often be implemented efficiently, they get complicated quickly, as even a simple application can have many states. In those cases where several things are happening at the same time, such as playing instructions to the user while listening for touch-tones, the complexity is compounded. The complex code required to manage all of the events, timeouts, and exceptions often obscures the primary intent of the application code.

PhoneScript Language Design

PhoneScript was created to provide a programming environment that makes writing interactive telephone applications easy to do. To achieve this end, the PhoneScript language was designed with several goals in mind. Simple applications should be short, and easy to write. More sophisticated applications should be possible, with the basic structure of their simpler cousins retained. Adding just one more feature to an application should not require a complete re-write of the code, just a minor addition. When the application is completed, the basic structure of the code should match its conceptual structure. One shouldn't have to be a contortionist to translate the application into the language. PhoneScript is a language intended for interactive applications. Each complete interaction, or "transaction" with the user, should be captured by a single language construct. The design of interactive applications is hard to get right the first time. Consequently its important that applications are easy to debug and modify, with an incremental style of application refinement encouraged. Finally, it should be easy to interface telephone applications to existing systems and applications, such as Fax, electronic mail, or graphical user interfaces.

PhoneScript consists of the 13 telephone interface commands listed in Table 1. These commands are used in conjunction with the built-in functions of TCL. I will not fully describe the TCL language here. Instead, I will note only the features of TCL required to follow the PhoneScript example programs. TCL provides the typical expression evaluation primitives, flow control constructs (such as *for, while, if-then-else* and *switch),* and procedures typically found in procedural languages. TCL operates on white space separated lists of ASCII character strings that are terminated by new lines or semicolons (;). The first string in a list is the command, with the remaining strings passed to the command as arguments. White space may be included in a string enclosing it in quotes ("), or by surrounding the string with braces ({}). The use of braces, which may be nested, also prevents variable and command substitution. Brackets ([]) are used for command substitution where *[command]* in TCL is analogous to *'command'* in the shell. The value of a variable is obtained by *$variable,* or *$variable(member)* for an array, where a backslash (\) can be used to prevent the special meaning of *$*. TCL also provides a wealth of built-in string and list manipulation commands. The PhoneScript functions in Table 1 are added to the core TCL commands to provide the telephone application specific capabilities of PhoneScript.

Name	Command Description
audio	Low level control of the audio system
beep	Play *beeping* tones
call	Place an outgoing phone call
cnv2tt	Convert an alphanumeric string to touch-tones
db	Structured file management commands
debug	Interactive debugging
hangup	Hangup the telephone line
on	event processing
phone	Direct phone line interface manipulation
play	Play audio files and receive touch-tones
prdate	Date and time conversion and formatting
record	Record an audio file
synth	Text to speech conversion

Table 1: Summary of PhoneScript Commands

PhoneScript uses the notions of event handling and implicit iteration to provide a framework for straight forward application development. Since PhoneScript is intended primarily for interactive telephone based applications, all of the setup and initialization of the telephone, audio, and DSP subsystems is taken care of automatically, with many configurable parameters set to useful default values.

As an interpreted language, PhoneStation simplifies program development by allowing interactive debugging of applications. The low level time

critical tasks are handled within compiled C code, so actions that happen at the interpreter level are *human response* kinds of actions. Several tenths of a second response time for their execution is not objectionable.

The PhoneScript main program manages most of the required book keeping. It initializes the telephone line interface, and the audio and text-to-speech sub-systems. Application programs use special global variables to customize the initialization of the system. The semantics of TCL are extended to permit command arguments of the form: *keyword=value*. When included as a command argument, they override the global value of the *keyword* parameter for the duration of the command. Applications can set useful default values at the top of the program, then override them on a command by command basis.

Sample PhoneScript Applications

The following PhoneScript examples, which are complete, working PhoneScript programs, will be used to illustrate the key features of the PhoneScript language. In the examples, items printed in `this font` represent PhoneScript code fragments or commands.

```
#!/usr/local/bin/PhoneScript
# place a call say: hello world

set usage "Usage: $argv(0) <number>"
if {$argc < 2} {
  puts stderr $usage; exit 0 }
synth "Hello World."
call $argv(1)
play # until .
exit 0
```

Figure 3: PhoneScript Version of **Hello World**

The first example is shown in Figure 3. This is the PhoneScript version of the *Hello World* program. The PhoneScript version synthesizes the phrase *Hello world*, places a phone call to the number specified on the command line, and speaks *hello world* when the called party answers the telephone. PhoneScript imports the command line arguments and the environment from the shell, so PhoneScript programs can be run directly from the shell. The `synth` command controls the text to speech synthesizer. It works in the background, leaving the resultant audio data on a queue when the synthesis is complete. The `call` command places the telephone call, and `play` sends the synthesized audio to the telephone line. The '#' instructs `play` to use the `synth` queue, instead of looking for a pre-recorded audio file. Although this example does quite a bit more than the standard C language version of *hello world*, it requires about the same amount of code. The setup required to operate the telephone line is handled automatically.

The second example, shown in Figure 4, is a simple, yet functional answering machine application. When the telephone rings, PhoneScript waits for 3 rings (the default), answers the telephone, then plays a pre-recorded greeting message. After the *beep* the caller can leave a message, which is saved as digitized audio in a UNIX file, and forwarded via electronic mail to the PhoneScript user. The remaining examples will build upon this one to enhance its functionality and to explore features of the PhoneScript language.

The variables `greeting`, `action`, and `timelimit` are initialized when the application begins. The `greeting` variable contains the outgoing greeting message, which can be recorded either by using another PhoneScript application, or with any of the standard audio applications that are available on the SPARCstation, such as *soundtool* [6]. The variable `action` contains the name of the UNIX command that will be invoked to deal with the message left by the caller. The digitized audio representation of the message is available as the standard input to that command. The `timelimit` variable is one of many PhoneScript configuration parameters. It sets the time to wait for the user to reply to a greeting message before proceeding to the next command. In this example, we with to start recording a voice message as soon as the greeting is finished playing, so the `timelimit` is set to zero.

```
set greeting $HOME/message.au
set action voice2mail
set timelimit 0

on call {
   set msg msg.[prdate].au
   exec touch $msg
   play $greeting until #
   beep
   record  $msg "" record_timeout=450
   }

on hangup {
   exec < $msg $action
   }
```

Figure 4: Complete Answering Machine Program

Unlike the *Hello World* example, where each statement is executed sequentially, the bulk of the work in the answering machine is done by the event handling constructs, `on call` and `on hangup`. PhoneScript waits for a phone call to come in, answers the telephone, then runs the body of the `on call` command. The PhoneScript `prdate` command returns the UNIX time, which is used to name the message file. The `play` command plays the prerecorded audio message. By default, `play` plays the audio message to completion. The `until` keyword specifies a regular expression that causes `play` to terminate immediately if the touch-tones keyed by

the user match the expression. In this case, keying the '#' key on the telephone keypad will cause the answering machine program to skip over the rest of the greeting, *beep*, then start recording. After the `on call` code is concluded, either because the caller hung up, or the message time limit was exceeded, PhoneScript hangs up the telephone, then runs the body of the `on hangup` command. The TCL built-in command `Exec`, calls `voice2mail`, a short shell script that converts the digitized voice message into a MIME format multi-media email message [7] by encoding it in ASCII, prepending the appropriate mail header lines, and forwarding it on to sendmail [8] for delivery. After the `on hangup` commands are finished, PhoneScript waits for the next call to arrive.

In PhoneScript applications that answer telephone calls, all but one time initialization code is in the body of one of the `on` event conditions, which are summarized in Table 2. The code associated with each event is read and saved during the initial scan of the PhoneScript program, but it is parsed and executed only when the corresponding condition occurs. This event handling mechanism in PhoneScript allows applications to deal with the asynchronous nature of the application domain in a straight forward manner.

Name	Event Description
on call	A telephone call is answered
on endringing	The telephone stopped ringing before the call was answered
on hangup	The telephone call was terminated
on int	The PhoneScript program was interrupted from the keyboard
on ringing	The telephone started to ring
on signal	The PhoneScript program was signaled by another process

Table 2: Summary of PhoneScript Event Conditions

One of the primary benefits of PhoneStation is its ability to use the telephone as simply another user interface to the workings of the computer. If the answering machine program is running all of the time, there needs to be a mechanism for escaping from the answering machine into more sophisticated telephone based applications. One way to accomplish this is to have the user key in as touch-tones a secret code while the answering machine is playing its greeting, a common technique used in consumer answering machines. In PhoneScript we can create any number of applications, and assign each one its own sequence of touch-tone codes. The name of each application will be the code needed to invoke it.

To accomplish this, the commands in Figure 4 are replaced by the code in Figure 5. The lines that have been emboldened mark the changes.

```
set greeting $HOME/message.au
set action voice2mail
set timelimit 0

on call {
  set msg msg.[prdate].au
  exec touch $msg
  play $greeting until # unless {
    if {$unless(tone) == "*"} hangup
    continue
    }
  catch {source $tones.tcl}
  beep
  record  $msg "" record_timeout=450
  }

on hangup {
  exec < $msg $action
  set tod [prdate "%A, %l %M %p."]
  synth $tod to $msg.tod
  }
```

Figure 5: Revised Play Command

Until now, `play` has been used to play an audio file and (optionally) stop after receiving a touch-tone. In the general case, a single `play` command can be used to support an entire transaction with the user, playing many audio files, and using touch-tones keyed by the user to guide the sequence in which the files are played. The `unless` option to `play` causes the TCL expression after the `unless` to be run any time a touch-tone is keyed by the user. While in the `unless` expression, a number of special PhoneScript variables that describe the current state of the `play` command are available, and can be examined or changed to customize the action of `play`. Using this technique, the special cases and exceptions can be handled from within a single `play` command, eliminating the need to bury a single user interface transaction in a maze of *ifs* and *elses* that would ordinarily be required to manage the special cases.

The TCL array `unless` contains a member for each of the variables passed by `play` to the `unless` expression. The just keyed touch-tone is stored in `unless(tone)`. and the accumulation of touch-tones keyed in so far is stored in `unless(tones)`. With this variation of the answering machine, when the user keys a `'*'` on the telephone keypad, the answering machine program executes a `hangup`, which immediately hangs-up the telephone line. This is invaluable in those cases where the answering machine picks up the call just as you are about to. When the greeting message is finished playing (or the user keyed a '#'), the variable `tones`, which is set by `play` as it finishes, contains the list of touch-tones entered

while the play command was running. Normally play will stop playing voice files whenever a touch-tone is entered, as this is usually the desired behavior. In this application, the continue command instructs play to continue playing the message file even though a tone has been received. However, the '#' tone will still skip the rest of the greeting message, and proceed directly to the *beep*.

After the play command is finished, the TCL source command runs the application program (if any) whose name matches the tones entered. If the user keys the touch-tones *123#*, the answering machine program will attempt to include the program 123#.tcl. The TCL catch command prevents the answering machine from flagging the error if the file *123#.tcl* does not exist.

Planning ahead for the next example, two additional commands are added to the on hangup expression, that will cause a *time of day* file to be created with each voice message. As before, prdate formats the current time and day, this time in a manner that can be easily read aloud. The argument to prdate calls the UNIX strftime() function, which replaces the *%X* constructs with the appropriate date and time strings. The Synth command converts the time and date string to speech, and saves it in a file. If the file is later played, it will say something like *Tuesday, eight forty-six PM.*

This example demonstrates the PhoneScript notion of implicit iteration. The behavior of the play command is guided by user input. As new features are added to the interaction, the additional functionality is expressed from within play, with out having to restructure the code. With this added functionality, the answering machine application functions as a gateway to many other applications. New features are added to the answering machine by creating the functionality as a separate PhoneScript program fragment. The user accesses the function simply by entering its name.

The next example, in Figure 6, a voice message browser named *123#.tcl*, is accessed from within the answering machine by entering the touch tones *123#* while the greeting message is playing. This example shows how a single play command can be used to manage a complex transaction with the user.

First we use the TCL builtin glob, that works like the *csh* command of the same name to create a list of the current voice mail messages. The files *intro.au* and *done.au* are pre-recorded messages, that contain the audio equivalent of *Playing voice mail messages* and *Done playing messages* respectively. The play command plays the introductory message, followed by the voice mail messages, then the concluding message in sequence. Touch tones keyed in by the user are used to alter the playback sequence, as controlled by the unless expression.

```
set msgs [glob "{intro,msg.*,done}.au"]
set reason=tf; set timelimit=0

play $msgs until "9" unless {
  case $unless(tone) in {
    "#" {incr unless(file); beep}
    "0" {play $unless(file_name).tod}
    "1" {incr unless(file) -1; beep}
    "*" {incr unless(file_pos) -16000}
    ""  {beep}
      }
   continue
   }
hangup
```

Figure 6: Program 123#.tcl – A Message Browser

The touch-tones '#', '0', '1', and '*' cause play to alter the default sequential playing of the messages. A '#' causes a skip to the next message, by incrementing the play variable *unless(file)*, the current file in the message list. A '0' causes play to chime in with the time and date that the voice message was recorded, by playing the *time of day* file that was created when the message was recorded. A '1' causes the playback to skip backward to the previous message. Finally, pressing '*' causes the previous two seconds of the message to be re-played, providing another opportunity to write down the phone number you missed the first time. The little details, such as trying to skip backward before the first message, are dealt with automatically by PhoneScript.

Normally the unless expression runs only when a touch tone is entered by the user. However the configuration variable reason is set to alter the conditions that cause unless to run. In this example, when a message is finished playing, and the next one is about to start, the unless expression is run. The last case of the case statement, for which there is no touch tone, is taken when one of the audio files finishes, causing a *beep*, informing the user that the current message file has finished playing. Additional features of the message browser, such as deleting messages, or forwarding them to other programs, are easily added within this framework, by adding new cases into the case statement. The continue statement prevents play from terminating when the first tone is entered.

Once the voice mail message browsing is complete, it is unlikely that returning to the answering machine program to record a voice message is still desired. The hangup command causes the message browser to hang up the phone at once, skipping the message taking part of the answering machine.

Although this answering machine does the job, the on ringing event of PhoneScript, activated just as the telephone begins to ring, enables an application to made decisions about a telephone call before the answering the telephone. For example, if Calling Number Delivery [9] (sometimes called

caller-id) is available, the TCL variable `number` contains the calling number when the `on ringing` section is run, so actions can be taken selectively based on the telephone number of the calling party. The code in Figure 7 is added to the answering machine program in Figure 5.

```
set caller_id 1
on ringing {
  if {[catch {source $number.tcl}]} {
    set greeting $HOME/message.au
    set action voice2mail
    set rings 3
    }
  }
```

Figure 7: Select Actions Based on Calling Number

The variable `caller_id` is set to turn on Calling Number Delivery, currently implemented by a readily available Calling Number Delivery interface, connected to the other serial port of the SPARCstation. When the telephone begins to ring, the `on ringing` code is executed before PhoneStation answers the call. As with the message browser in the previous example, if a file name matches the calling number, its contents are read and executed as part of the application. If no file exists, the greeting and action are reset to their default values. The variable `rings` is the count of rings to wait before PhoneScript answers the telephone. By creating a file whose name is the telephone number of the boss, a special message is played only when the boss calls. The contents of that file might contain:

```
set rings 1
set greeting boss.au
set action "page_me 'The boss called'"
```

If it is the telephone number for the collection agency instead, the file might contain:

```
set rings 99
```

even they don't have that much patience.

If Calling Name delivery is not available, the answering machine can still be programmed to choose different messages. This time the answering machine will be coupled with a configuration file to allow the greeting message and number of rings to wait before picking up the call to be chosen, based on the time of day and the day of the week. For example, the caller can be made to wait for 3 rings and be greeted with *good morning* on Tuesday mornings. If answering the telephone is not desired, PhoneStation can pick up the telephone at the first hint of ringing to play an appropriate message.

This feature is implemented in PhoneScript using a structured file. An example of which is shown in Figure 8. A structured file in PhoneScript consists of 1 or more line of text, each containing semi-colon terminated fields. The first line in the file names the fields, whose values are accessed via TCL variables of the same name. The remaining lines are the data. This configuration file has six fields. The first, *days* contains a range of days, 0 for Sunday, 1 for Monday etc. The next two fields give a range of times, in military time, for which this line applies. The fourth and fifth fields give the names of two pre-recorded message files that are played consecutively as the greeting message. The first message is used for a salutation, such as *good morning* and the other one for instructions, such as *Please leave a message at the beep.* The final field specifies the number of rings to wait before picking up the telephone.

```
days;start;end;greeting;message;no_rings;
0-6;630;1200;morning.au;;;
0-6;1200;1630;afternoon.au;;;
0-6;1630;1830;evening.au;;;
1-5;630;830;;;5;
1-5;830;1630;;work.au;2;
06;900;2100;;weekend.au;4;
0-6;0;2400;off_hours.au;default.au;1;
```

Figure 8: Greeting Message Configuration File

This structured file is accessed through the PhoneScript `db` command, by including the code from Figure 9 into the answering machine program in Figure 5 instead of the Calling Name Delivery code.

```
on ringing {
  set day [prdate %w]
  set hour [prdate %k%M]
  set msg1 ""; set msg2 ""; set rings ""
  db select {[string match \[$days\] $day]}
  db select and "\$end > $hour"
  db select and "\$start <= $hour"
  db process {
    if {$msg1$greeting == $greeting} {
      set msg1 $greeting }
    if {$msg2$message == $message} {
      set msg2 $message }
    if {$rings$no_rings == $no_rings} {
      set rings $no_rings }
  }
  set greeting "$msg1 $msg2"
```

Figure 9: Greeting Message Selection

The plan is to choose two different greeting messages, to be played consecutively, and the number of rings to wait until the telephone is answered. The first two set commands figure out the current day of the week and hour of the day. The variables `msg1` and *msg2,* which will contain the two greeting messages, start off empty, as will `rings`. The `db select` command evaluates its argument as a TCL expression for each line of the configuration file, with the TCL variables corresponding to each field name containing the value for the current row. Only those rows for which the expression is true remain selected. After the three `db select` commands are finished, only those rows in the database that match the current time and day will be selected.

The code in the `db process` command gets executed once for each selected row in the database. The first selected row in which either of the messages or the number of rings is specified, causes the appropriate value to be filled in. The final `set` command sets the greeting message to the concatenation of the two message files. The message files contain pre-recorded messages.

```
# Interactively edit a procedure
proc editproc {name} {
  global pid argv
  if {[info procs $name] == ""} {
     echo "$name not found"; return }
  set file /tmp/$name.$pid.tcl
  set fd [open $file "w"]
  set args [info args $name]
  set body [info body $name]
  puts $fd "# $argv(0) [prdate {%D %T}]\n"
  puts $fd "proc $name \{$args\} \{$body\}"
  close $fd
  exec vi $file < /dev/tty > /dev/tty
  uplevel "source $file"
  }
```

Figure 10: Interactively Edit a PhoneScript Procedure

The sample applications shown so far have been simple, and chances are good that they could be typed in and work on the first try. More complex applications can be debugged interactively using the built-in debugging features of PhoneScript. PhoneScript is normally run in batch mode, by running an existing PhoneScript program. PhoneScript may also be run interactively, like the *shell*. The user is prompted for commands from the keyboard. This is a useful way to test fragments of an application. This can be a tedious way to develop entire applications, however. The PhoneScript command `debug` causes PhoneScript to enter interactive mode from within a batch file, accepting TCL and PhoneScript commands from the keyboard. If the TCL *variable* `debug` is set, then PhoneScript will automatically enter interactive mode when a PhoneScript command fails. The error can be corrected interactively by retyping the command, and batch mode resumed by typing `exit` from the keyboard. The use of `debug` can be further enhanced with a TCL procedure such as TCL procedure `edit_proc`, shown in Figure 10, that can invoked interactively with the name of a (presumably errant) procedure. The `edit_proc` procedure writes the procedure provided as an argument into a file, starts up a text editor with that file, then reads the procedure back into the running PhoneScript program. Using this facility, the core of an application can be written in advance, and the remainder while the application is running. A missing feature will cause an error, interactive mode will begin, the new feature can be added, and execution of the program resumed.

As a final debugging aid, each PhoneScript command is assigned a letter that causes it to display various diagnostic and debugging information, when that letter is contained in the value of the *debug* variable. The various types of diagnostics may be enabled or disabled simply by changing the value of `debug`.

Related Work

The BerTel computer controlled telephone switch [10, 11] demonstrated that telephone and computers can talk to each other. The system also pointed out there needs to be a better way of constructing new telephone based services. The Expect [12] language shows how interactive programs can be tied together with a procedural language that has a built in notion of timeouts as expected conditions. The TCL embeddable command interpreter proved to be easy enough to use, that its simpler to build the right tool for a particular task, than it is to force the wrong one into service. Finally, the availability of multi-media mail transport facilities [13] and multimedia email user interfaces [14] provide PhoneStation with a connection into the workstation environment.

Summary and Conclusions

In addition to assorted answering machine programs, PhoneStation has been used to construct a directory assistance service, a survey system, an automatic scheduling program, and a fax document server. The survey system, used to evaluate the quality of the ORATOR® speech synthesizer under varying speaking parameters [15] was constructed in PhoneScript by a summer student with no prior UNIX experience in a couple of weeks. PhoneStation is in continuous service as part of the multimedia email system, providing users without audio capabilities on their workstations the ability to generate audio email, and to receive the audio portions of multimedia email messages over the telephone.

The ease of incorporation of TCL into the PhoneStation environment for the creation of PhoneScript is a tribute to the design of TCL. New flow control constructs, such as the PhoneScript event handling, and the extension of the `continue` semantics within the `play` command were easy to implement, eliminating the need to build a special purpose command interpreter for PhoneStation. As new technologies become available, such as speech recognition, new PhoneScript commands can be added to extend its functionality while maintaining the existing framework. Several applications, including the automated directory assistance system, were written twice, once in C using the library interface, and again directly in PhoneScript. In all cases the PhoneScript applications were shorter, easier to write, and took less time to get working than the C

language versions. The interactive response of both versions is essentially the same.

PhoneStation demonstrates that the telephone, which has been traditionally ignored as a component of a workstation environment, can be integrated successfully, and provides not only better control of the telephone than an ordinary telephone, but extends the capabilities of the workstation as well.

References

[1] Goldberg, A., (ed) *A History of Personal Workstations* ACM Press, 1988 pp 316.

[2] ZiLOG. *Z8 Family Design Handbook* Campbell Ca., 1989

[3] Ousterhout, J. *TCL: An Embedded Command Language* USENIX Winter conference proceedings, January, 1990, pp 133-146.

[4] Spiegel, M.F., Macchi, M.J., and Gollhardt, K.D. *Synthesis of names by a demisyllable-based speech synthesizer (ORATOR®),* Eurospeech '89 Conference Proceedings, September 26-28, 1989, pp 117-120.

[5] Kaiser, J. *Algorithms for Second Order Recursive Digital Filter Design* Unpublished Memorandum, 1992.

[6] Sun Microsystems *Soundtool Manual Page* SunOS Reference Manual March 1990, pp 1782-1784.

[7] Borenstein, N., and Freed, N. *MIME (Multipurpose Internet Mail Extensions): Mechanisms for Specifying and Describing the Format of Internet Message Bodies* RFC 1341, Internet Advisory Board, June, 1992.]

[8] Sun Microsystems *Sendmail Manual Page* SunOS Reference Manual March 1990, pp 2100-2102.

[9] Bellcore *CLASS Feature: Calling Number Delivery* Bellcore Technical Reference TR-TSY-000031 Issue 3, January, 1990

[10] Redman, B. *Who Answers Your Telephone When You're in the Information Age?* Summer 1985 USENIX Conference Proceedings Portland, OR, pp 569-576.

[11] Redman, B. *A User Programmable Telephone Switch* Unpublished internal Bellcore memorandum, April, 1988.

[12] Libes, D. *Expect, Curing those Uncontrollable Fits of Interaction* USENIX Summer conference proceedings, June 1990, pp 11-15.

[13] Borenstein, N. *Multi-media mail from the bottom up or Teaching Dumb Mailers to Sing* Winter Usenix Conference proceedings, January, 1992, pp 79-91.

[14] Uhler, S. *MUI, a Window Based User Interface for Multi-Media Mail* Proceedings of the Bellcore/BCC Symposium on User Centered Design, Bellcore Special Report SR-OPY-002130, November, 1991, pp 171-175.

[15] Macchi, M. J. et al *Intelligibility and Naturalness as a Function of Speaking Rate and Word Boundary Strength in the ORATOR® System* Presented as a talk at the IEEE Workshop on Interactive Voice Technology for Telecommunications Applications. October 19, 1992

Author Information

Stephen Uhler joined Bell Communications Research at its inception in 1984, where he is a Member of the Technical Staff in the Computer Systems Research division. He has worked on computing environments and user interfaces for much of that time, and is the author of the MGR window system. Before joining Bellcore, Stephen was a Member of the Technical Staff at Bell Laboratories in Whippany N.J. where he worked on user interface management systems. He received an M.S. degree from Case Western Reserve University. Stephen can be reached via electronic mail at: sau@bellcore.com or uunet!bellcore!sau.

Glish: A User-Level Software Bus for Loosely-Coupled Distributed Systems

Vern Paxson & Chris Saltmarsh – Lawrence Berkeley Laboratory

ABSTRACT

We describe *Glish*, an interpreted language for building distributed systems from modular, event-oriented programs. These programs are written in conventional languages such as C, C++, or FORTRAN. Glish scripts can create local and remote processes and control their communication. Glish also provides a full, array-oriented programming language for manipulating binary data sent between the processes. In general Glish uses a centralized communication model where interprocess communication passes through the Glish interpreter, allowing dynamic modification and rerouting of data values, but Glish also supports point-to-point links between processes when necessary for high performance. Glish is available via anonymous ftp.

1 Introduction

Much of the power of Unix stems from the ways in which users can combine different programs. The notions of standard input and output, pipelines, filter programs, and command shells all encourage the creation and use of modular programs that can be "plugged together" in novel ways. Traditionally Unix command shells have focussed on creating and connecting together processes. Recently, however, command shells such as *perl* [14] also provide powerful languages for manipulating the output generated by programs. Often a *perl* user can write a considerable portion of a task in *perl*, rather than needing to create new filter programs. We might say that in this regard *perl* provides better "glue" than previous shells for connecting together programs.

There are some limits, however, to the power of Unix pipelines, even when augmented with a shell like *perl*. Data in pipelines flows in only one direction; two programs cannot communicate with each other back and forth. Furthermore, the data the programs manipulate is generally limited to character streams whose structure is column- or line-oriented. Communicating large quantities of numeric data is inefficient at best and inaccurate unless care is taken. Communicating *structured* data – collections of related values, perhaps with different types – is particularly difficult.

While it is possible to circumvent these restrictions, the only support for doing so is at the operating-system-call and run-time-library level. There is no analog of shell programming for interconnecting processes so they can communicate in complex ways and share binary, typed data.

Applications such as simulation systems often can be well modeled as a number of separate processes perhaps running on different hosts that occasionally send structured data back and forth; i.e., as loosely-coupled distributed systems. Since the present facilities in Unix provide little high-level support for such an approach, one instead often resorts to writing the system as a set of processes that have considerable knowledge about what other processes and data structures exist in the system. This system-specific knowledge makes it difficult to extend the system in unforeseen ways, so unless one has a complete understanding of the system requirements at the outset, one is likely to find the final system uncomfortably restrictive.

In this paper[1] we discuss a *software bus*-style solution to building flexible, loosely-coupled distributed systems. The main thrust of the software bus approach is that individual programs should be wholly modular, with no knowledge of other programs or data types that might exist in the system. The software bus supplies a uniform way for programs to communicate without knowing about one another. In our system, programs are written in terms of *events*, which are name/value pairs. In the usual case, programs receive an event, perform some sort of action in response to the event, and possibly generate one or more new events associated with the response. Such programs are similar to RPC servers, except that "calls" to the programs are not synchronous. An example is an FFT server, which might be sent an event with the name "please-FFT-this" and an associated value of an array of double precision data, to which the server in turn generates an "FFT-done" event whose value is two arrays, the

[1]Work supported by the U.S. Department of Energy under Contract No. DE-AC03-76SF00098. The second author is with the Superconducting Super Collider Laboratory, operated by the Universities Research Association, Inc., for the U.S. Department of Energy under Contract No. DE-AC02-89ER40486.

Fourier components of the original data. More generally, programs can also spontaneously create events in response to external actions, such as a piece of hardware signaling that some condition has changed, a timer going off, or a person interacting with a graphical interface.

Our software bus, called *Glish*, has three parts:

- a C++ class library that programs (Glish *clients*) link with so they can generate and receive events and manipulate structured data;
- the Glish "sequencing" language analogous to *perl* (but considerably different in flavor);
- an interpreter process for executing Glish scripts and acting as a central "clearing-house" for forwarding events between processes.

The Glish system is very flexible:

- existing programs can be turned into Glish clients either by writing event-oriented, C++ "wrappers" around them or by encapsulating their filter behavior using `stdin` and `stdout` events;
- clients in a Glish script can run on different computers, which can have heterogeneous architectures;
- Glish provides a full programming language for manipulating the events and data generated by and sent to clients.

In the next section we present an example of the type of systems we want to build with Glish, show how we would use Glish to construct the system, and then present several refinements to convey the flavor of the Glish approach. In Section 3 we give an overview of the more conventional aspects of the Glish language and in the following section discuss those facets of the language concerning event-oriented interprocess communication.

In Section 5 we discuss the C++ class libraries used to integrate programs into the Glish system and give an example of an "FFT" server written using the libraries. The next two sections discuss the implementation and performance of the system. We then conclude with an overview of related work, the present status of the system, and our thoughts on future work.

2 Example of Building a System Using Glish

For an idea of the sorts of problems Glish is meant for and how it's used to solve them, consider a simple example where we want to repeatedly view readings generated by an instrument attached to a remote computer called "mon". Suppose we have a program *measure* that reads values from the special hardware device and converts them into two floating-point arrays, `x` and `y`. *measure* needs to run on the remote host "mon" because that's where the special hardware resides. We have another program, *display*, for plotting the x/y data, which we want to run on our local workstation. *display* also has a "Take Measurements" button that we can click on to instruct the hardware to take a new set of measurements.

The first problem we're interested in is simply to connect together *measure* and *display* so that when *measure* produces new values they're shown by *display*, and when we click the display's button *measure* goes off and reads new values. Figure 1 illustrates the flow of control and data: *display* tells *measure* to take measurements, and *measure* informs display when new measurements are available.

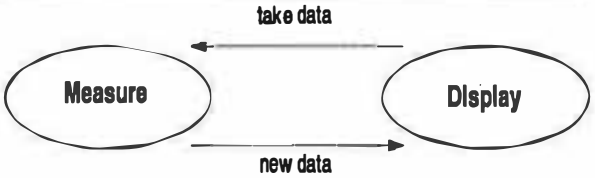


Figure 1: Simple Two-Program Distributed System

To implement even this simple system under Unix requires constructing a session-layer protocol which then has to be implemented on top of sockets or RPC. When using Glish, though, the protocol and the communication mechanism are built-in. Every program in a Glish system communicates by generating *events*, messages with a name and a value. For our simple system we might write *measure* so that whenever it has new readings available it generates an event called `"new_data"`. The value of the event will be a record with two elements, `x` and `y`, the two arrays of numbers it has computed from the raw measurements. We would write *display* so that when it receives a `new_data` event it expects the value of the event to be a record with at least `x` and `y` fields; it then plots those values. Similarly, when we push the "Take Measurements" button *display* will generate a `take_data` event, and whenever *measure* receives a `take_data` event it will get a new set of readings and generate a new `new_data` event.

Here is a Glish script that when executed creates the two processes, one remotely, and conveys their messages to each other:

```
m := client("measure", host="mon")
d := client("display")

whenever m->new_data do
    d->new_data( $value )

whenever d->take_data do
    m->take_data( $value )
```

When Glish executes the first two lines of this script it creates instances of *measure* (running on the host "mon") and *display* (running locally) and assigns to the variables `m` and `d` values corresponding to these Glish clients. Executing the next line:

```
whenever m->new_data do
```

specifies that whenever the client associated with `m`

generates a `new_data` event, execute the following statement:

```
d->new_data( $value )
```

This statement says to send a new event to the client associated with `d`. The event's name will be `new_data` and the event's value is specified by whatever comes inside the parentheses; in this case, the special expression `$value`, indicating the value of the most recently received event (*measure*'s `new_data` event).

The last two lines of the script are analogous; they say that whenever *display* generates a `take_data` event an event with the same name and value should be sent to *measure*.

Our system could easily be a bit more complicated. Suppose that prior to viewing the measurements with *display* we first want to perform some transformation on them. The transformation might for example calibrate the values and scale them into different units, filter out part of the values, or FFT the values to convert them into frequency spectra. Rather than building the transformation into *measure*, we would like our system to be modular, so we use a separate program called *transform*.

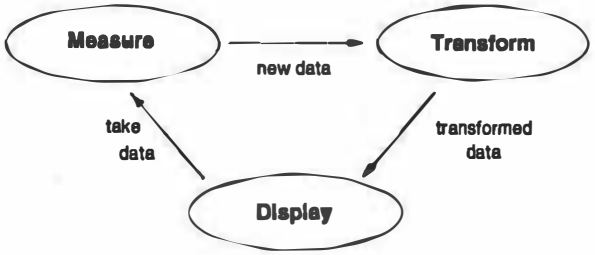


Figure 2: Three-Program Distributed System

Figure 2 shows the flow of control and data in this new system. *measure* sends its values to *transform*; *transform* derives some transformed values and sends them to *display*; and *display* tells *measure* when to take more measurements. With Glish it's easy to accommodate this change:

```
m := client("measure", host="mon")
d := client("display")
t := client("transform")

whenever m->new_data do
    t->new_data( $value )

whenever t->transformed_data do
    d->new_data( $value )

whenever d->take_data do
    m->take_data( $value )
```

The third line runs *transform* on the local host and assigns a corresponding value to the variable `t`. The first `whenever` forwards `new_data` events from *measure* to *transform*; the second `whenever` statement effectively forwards *transform*'s `transformed_data` events to *display*, but changes the event name to `new_data`, since that's what *display* expects. The third `whenever` is the same as before.

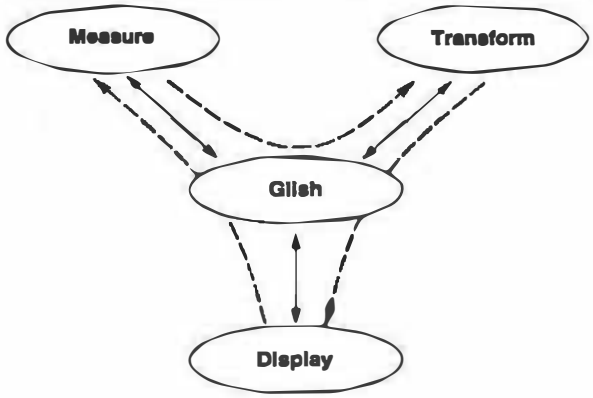


Figure 3: Conceptual Event Flows vs. Actual Flows

An important point in this example is that while *conceptually* control and data flow directly from one program to another, in reality all events pass through the Glish interpreter. Figure 3 illustrates the difference. Here solid lines show the paths by which events actually travel, while dashed lines indicate the conceptual flow. While this centralized architecture doubles the cost of simple "point-to-point" communication, it buys enormous flexibility. For example, suppose sometimes we want to use *transform* before viewing the data and other times we don't. We add to *display* another button that lets us choose between the two. It generates a `set_transform` event with a boolean value. If the value is true then we first pass the measurements through *transform*, otherwise we don't.

To accommodate this change in our Glish program we could add a global variable `do_transform` to control whether or not we use *transform*:

```
m := client("measure", host="mon")
t := client("transform")
d := client("display")
do_transform := T

whenever m->new_data do
    {
    if ( do_transform )
        t->new_data( $value )
    else
        d->new_data( $value )
    }

whenever t->transformed_data do
    d->new_data( $value )

whenever d->take_data do
    m->take_data( $value )

whenever d->set_transform do
    do_transform := $value
```

We initialize `do_transform` to `T`, the boolean "true" constant. We change it whenever *display* generates a `set_transform` event (see the last two lines). When *measure* generates a `new_data` event we test the variable to determine whether to pass the event's value along to *transform* or directly to *display*.

Furthermore, if the data transformation done by *transform* is fairly simple, we could skip writing a program to do the work and instead just use Glish. For example, suppose the transformation is to find all of the `x` measurements that are larger than some threshold, and then to set those `x` measurements to the threshold value and the corresponding `y` measurements to 0. We could do the transformation in Glish using:

```
m := client("measure", host="mon")
d := client("display")
do_transform := T

if ( len(argv) > 0 )
    thresh := as_double(argv[1])
else
    thresh := 1e6

whenever m->new_data do
    {
    if ( do_transform )
        {
        too_big := $value.x > thresh
        $value.x[too_big] := thresh
        $value.y[too_big] := 0
        }

    d->new_data( $value )
    }

whenever d->take_data do
    m->take_data( $value )

whenever d->set_transform do
    do_transform := $value
```

Here we first check to see whether any arguments were passed to the Glish script and if so we initialize `thresh` to be the first argument interpreted as a double precision value. If no arguments were given then we use a default value of one million.

Now whenever *measure* generates a `new_data` event and we want to do the transformation, we set `too_big` to a boolean mask selecting those `x` elements that were larger than `thresh`. We then set those `x` elements to the threshold, zero the corresponding `y` elements, and pass the result to *display* as a `new_data` event. We have eliminated the need for *transform*.

Finally, for situations in which performance is vital Glish provides point-to-point links between programs. The `link` statement connects events generated by one program directly to another program. The `unlink` statement suspends such a link (further events are sent to the central Glish interpreter) until another `link`. Here is the last example written to use point-to-point links:

```
m := client("measure", host="mon")
d := client("display")

link m->new_data to d->new_data

if ( len(argv) > 0 )
    thresh := as_double(argv[1])
else
    thresh := 1e6

whenever m->new_data do
    {
    too_big := $value.x > thresh
    $value.x[too_big] := thresh
    $value.y[too_big] := 0
    d->new_data( $value )
    }

whenever d->take_data do
    m->take_data( $value )

whenever d->set_transform do
  {
  if ( $value )
    unlink m->new_data to d->new_data
  else
    link m->new_data to d->new_data
  }
```

We now no longer need the `do_transform` variable. Instead we initially create a link for *measure*'s `new_data` events directly to *display*. Whenever *display* sends a `set_transform` event requesting that the transformation be activated, we break the direct link between *measure* and *display*. Now when *measure* generates `new_data` events they will be sent to Glish, which will then transform the data and pass it along to *display*.

These examples illustrate the main goals of Glish: making it easy to dynamically connect together processes in a distributed system, and providing powerful ways to manipulate the data sent between the processes. One other important point is that because *measure*, *transform*, and *display* are all written in an event-driven style, each of them can be easily replaced by a different program that has the same "event interface". For our own work (scientific programming) we often want to replace *measure* with *simulate* (a program that simulates the quantity being measured), *display* with a non-interactive program once we have ironed out the measurement cycle, and *transform* with a variety of different transformations. We also might want to run *measure* and *simulate* together, so we can compare *simulate*'s model with the actual phenomenon measured by *measure*. The ability to quickly "plug in" different programs in this fashion is one of Glish's main benefits.

3 The Glish Language

Overview

The design of the Glish language was heavily influenced by the *S* language [1]. Every value is a dynamically-typed array. The *S* types included are *numerics* (boolean, integer, float, and double, all of which can be freely mixed and coerced to one another), strings, functions, and records. Record fields can be accessed using string-valued expressions as well as with the field-name operator, so records provide a form of associative array.

We added two more types: *references* to other values (for efficiently dealing with large arrays) and *agents*, which are event producer/consumers. Agents typically are programs that have been linked with the Glish Client library (in which case they are called *clients*). They can also be shell commands or Glish "subsequences", similar to Glish functions.

Two levels of scoping are provided for variables, global and local to a function. Variables needn't be declared, except to explicitly set their scope. There is no "main" function; statements outside the scope of any function are executed when the Glish script begins. Here, for example, is the Glish "hello, world" program:

```
print "hello, world"
```

The usual control constructs are provided, along with five additional types of statements:

- *event-send* statements for sending events;
- `whenever` statements for specifying what should happen when an event is generated;
- `await` statements for synchronous communication;
- `link` statements for creating point-to-point communication links;
- `unlink` statements for suspending point-to-point links.

These are discussed in Section 4, below.

Glish also provides a number of predefined functions (such as `sqrt`, `max`, `sum`, all array-oriented) and variables. Examples of predefined variables are `argv`, the argument list with which the script was run, and `environ`, a record of the environment variables. For example, the current user name can be accessed using

```
environ["USER"]   or   environ.USER
```

Arrays

Most Glish types correspond to an *array* of values rather than a single value. For example,

```
a := [1, 2, 6]
b := [3, 4, 5]
print a + b
```

assigns two three-element integer arrays to `a` and `b`, and then prints their element-wise sum: `[4, 6, 11]`. You can also mix arrays and *scalars* (single-element arrays) in expressions:

```
print a * 2
```

will print `[2, 4, 12]`. Glish provides the usual arithmetic and logical operators; all operate element-by-element on two arrays of the same size, or, if one of the operands is a scalar, apply the scalar value to each element in turn.

Arrays automatically grow when you assign to an element beyond their current end. Given `a` as above, executing:

```
a[5] := 4
```

results in `a` having the value `[1, 2, 6, 0, 4]`.

Integer arrays can also be created using the built-in "`:`" operator, which returns an array of the integers between its operands. For example,

```
3:7
```

is equivalent to

```
[3, 4, 5, 6, 7]
```

Create string arrays by enclosing text within double quotes. The text is broken into words at each occurrence of white space, which is then discarded:

```
c := "hello, world"
```

assigns to `c` a two-element string array, the first element being "`hello,`" and the second element "`world`". Text enclosed in single quotes is treated as a string scalar:

```
d := 'hello, world'
```

assigns to `d` a single-element string value, with the white space preserved.

The `length` function returns the length of an array. It can be abbreviated as `len`.

Records

You can package together a collection of values into a *record*:

```
r := [a="hello", b=11:20]
```

assigns to `r` a record with two fields: `r.a` designates the scalar string `"hello"`, while `r.b` designates a ten-element integer array. You can also create records by directly assigning to a field:

```
s.constants := [3.14159, 2.71828]
```

creates a new record `s` and initializes its `constants` field to an array of two double-precision values.

Besides using the "`.`" operator, you can also access record fields using string-valued array indices:

```
print s["constants"][2]
```

prints `2.71828`. Record fields can also be referred to using integer indices:

```
print r[2]
```

prints the integers from 11 to 20.

Multi-Element Indexing

Glish provides ways for accessing or modifying more than one array element (or record field) at a time. For example, you can use an integer array as an index into another array:

```
a := [9, -3, 0, 7, 5]
b := [4,2]
print a[b]
```

prints `[7, -3]`. Since the "`:`" operator yields an integer array, you can use it to access a contiguous sequence of elements in an array: `a[3:5]` yields `[0, 7, 5]`.

You can use a boolean array as a *mask* for selecting which elements you want from the array:

```
print x[x >= 4 & x <= 12]
```

prints all the elements of `x` with values between 4 and 12.

Both integer and boolean array indices can also be used to assign to a subset of an array's elements:

```
x[x < 0] := -x[x < 0]
```

negates all of the negative elements in `x`, and

```
rev_x := x[len(x):1]
```

creates in `rev_x` a copy of `x` with the elements in reverse order.

You can select a subset of a record's field in a similar fashion:

```
r := [a=1, b="hi", c=9.3]
s := r["b c"]
```

assigns to `s` a record with two fields, the `b` and `c` fields of `r`.

References

A reference is a mechanism for two variables to share the same storage for their values. References are created using the `ref` or `const` operators. You can use `ref` references to both access and modify the variable; with `const` references you can only access the variable.

For example,

```
a := 1:5
b := ref a
b[2] := 9
print a
```

prints `[1 9 3 4 5]`.

An important point, though, is that while `a` and `b` refer to the same underlying storage, assigning either of them to another value breaks the connection between the two. If we do:

```
a := 1:5
```

then `a` will go back to equaling `[1 2 3 4 5]` while `b` will remain equal to `[1 9 3 4 5]`.

The reference connection can be maintained by explicitly stating that you want to do so by using the `val` operator. For example, after executing:

```
c := [1, 3, 7, 12]
d := ref c
val c := "hello there"
```

the value of `d` (and of course `c`) will be the two-element string `"hello there"`.

Functions

Glish provides a flexible mechanism for defining and calling functions. These functions are a data type; they can be assigned to variables or record fields, passed as arguments to other functions, and returned as results of functions. A function body can be either an expression or a block of statements. Here's a simple example of a function that prints its arguments and then returns their difference:

```
function diff(a, b)
    {
    print "a =", a
    print "b =", b
    return a - b
    }
```

You can make arguments optional by specifying default values for them. If in the above example we replaced the first line with:

```
function diff(a, b=1)
```

then when `diff` is called with only one argument `b` will be set to 1. So the call `diff(3, 7)` returns `-4`, and the call `diff(3)` returns `2`.

In a function call you can also give the function arguments by name instead of positionally:

```
diff(b=4, a=7)
```

returns 3, since 7-4=3.

The function definition above assigns a function value to the global variable `diff`. Functions can also be assigned to local variables and record fields:

```
data.transform :=
    function(x) log(x)/log(2)
```

assigns to the `data` record's `transform` field a function that returns log2 of its argument.

Arguments to Glish functions can be passed by value, by reference, or by `const` reference (the default), by preceding the argument's name in the function definition with `val`, `ref`, or `const`. Passing by reference allows Glish functions to deal with large values efficiently. Glish also supports variable argument lists, which are useful for writing "wrapper" functions that call other functions. For example,

```
function psych_client(...)
    client(..., host="psychosis")
```

defines a function that when called creates a Glish client on the remote host "psychosis".

One particularly useful predefined function is `shell`, which interprets its arguments as a Bourne shell command line and returns the output from running the command (optionally on a remote host) as a string value. For example,

```
csh_man := shell( "man csh" )
```

assigns to the variable `csh_man` a string array, each element corresponding to one line of the "csh" manual page, and

```
function to_lower(x)
    shell("tr A-Z a-z", input=x,
          host="cruncher")
```

returns its argument converted to lower-case, doing the work on the remote host "cruncher".

The `function` keyword can be abbreviated as `func`.

4 Events and Agents

Glish's main purpose is to coordinate a number of processes that form a distributed system. These processes are instances of programs written in compiled languages such as C or C++.

Each program is written in an *event-oriented* style; the program's sole view of the rest of the system comes from the *events* it receives, and its sole mechanism for communicating its state and results to the system is by generating more events. The programs have no knowledge of what other programs the system includes, or what is done with their results, or where received events came from. The *event-oriented* style lends itself to creating modular programs that you can connect together in novel ways. You make these connections using Glish.

We deal with the details of how programs themselves receive, interpret, and generate events later in Section 5. Here we focus on manipulating events from within a Glish program.

What is an "Event"?

An *event* has a *name* and an associated *value*. The name is simply an identifier, much like a variable's name. The value can be any Glish value, of any type: numeric, string, record, reference, agent, or function. We might speak of "a `foo` event with value `[3, 2, 5]`" to mean an event whose name is "foo" and value the three-element integer array `[3, 2, 5]`.

Agents

An *agent* is an entity that generates and responds to events. Typically it's a process running either locally or on a remote computer; these agents are called *clients*.

Agents generate events in order to communicate with the rest of the world, namely the Glish script and any other agents the script may have created. By saying that agents *respond* to events we mean that they expect to receive events with certain names, and when they do they perform some action based on the name and value of the event. The action may entail generating one or more new events or may not.

Glish predefines several events for every agent: `established` is generated when an agent first begins running; `unrecognized` is generated when an agent does not recognize an event sent to it; `done` is generated when the agent finishes successfully; `fail` is generated on behalf of an agent that terminates abnormally (e.g., due to a bus error); and `terminate` can be sent to any agent to tell it to exit. These events form the mechanism by which agents are controlled and errors detected.

Sending Events to Agents

Suppose that `a` is a Glish variable with an *agent* value. You can send an event to `a`'s agent using the `->` operator. Executing:

```
a := client("demo")
a->foo( [1, 4, 6] )
```

first associates `a` with an instance of the program *demo* running on the local host, and then sends a `foo` event to `a`'s agent (i.e., *demo*) with a value of `[1, 4, 6]`.

Sending an event is in some ways similar to making a function call. In particular, we can send more than one value:

```
a->foo( "value1", 2 )
```

sends an event with two values, the string `"value1"` and the integer `2`. The values can also be named:

```
a->foo( x="xval", y=5 )
```

sends an event with the "parameter" `x` equal to `"xval"` and `y` equal to `5`. Multi-valued events are equivalent to passing a single-valued event where the value is a record. This last example is equivalent to:

```
a->foo( [x="xval", y=5] )
```

The event name in a `->` operation needn't be fixed in advanced. Instead you can use any string-valued expression by enclosing it within brackets (`[ ]`'s). The following are equivalent:

```
a->foo( 5 )
a->["foo"]( 5 )
```

and here is one way to send `a` three events, `foo`, `bar` and `bletch`, with values of `1`, `2`, and `3`:

```
for ( i in 1:3 )
    a->["foo bar bletch"[i]]( i )
```

(Recall that `"foo bar bletch"` is a three-element array of strings.)

One major difference between sending an event and calling a function is that sending an event is an *asynchronous* operation. As soon as Glish has sent the event it proceeds to execute the next statement in

the Glish script. Events can be sent synchronously using the `await` statement, which we discuss in the "Receiving Events Synchronously" section, below.

Receiving Events from Agents

Again, suppose that `a` is a variable with an *agent* value. In a Glish program you can respond to events that `a` generates using a `whenever` statement. Once executed,

```
a := client("demo")
whenever a->bar do
    print "got a bar event"
```

will print `"got a bar event"` every time *demo* generates a `bar` event.

The value of the most recently received event is kept in a special variable `$value`:

```
whenever a->bar do
    print "got bar =", $value
```

will display the value of each `bar` event that `a` generates.

Agent values are also records, and the most recent value of each event is available as a field in the record. For example, the `print` statement in the `whenever` above could also have been written this way:

```
print "got bar =", a.bar
```

The value persists in `a.bar` until `a` generates another `bar` event, at which point `a.bar` is updated to reflect the new event's value.

Just as when sending events you can use a string-valued expression to name an event, so can you with `whenever`:

```
whenever a->["foo bar bletch"] do
    print $value
```

will print the value of each `foo`, `bar`, and `bletch` event generated by the agent `a`.

Finally, "`*`" can be used to indicate *every* event:

```
whenever a->* do
    print $value
```

prints the value of every event `a` generates.

Along with `$value`, two other special variables are available in the body of a `whenever`: `$name` holds the name of the event and `$agent` is a reference to the agent that generated it. For example, the following function:

```
function setup_relay(src, ref dest)
    {
    whenever src->* do
        dest->[$name]($value)
    }
```

executes a `whenever` statement relaying every event generated by the agent `src` to the agent `dest`. (`dest` has to be declared a `ref` parameter since sending an event to an agent is considered to modify the agent.) Note that the `whenever` statement "persists" even after a call to `setup_relay` returns.

There is no restriction on the body of a `whenever`. It can include function calls, agent creation, and further `whenever` statements, for example.

Receiving Events Synchronously

An `await` statement instructs Glish to wait for an event to occur. Glish pauses program execution until this happens. For example, suppose that `c` refers to a client that when sent a `compute` request performs some computation and generates a `compute_done` event when finished. If after you tell `c`'s client to do its computation you want to wait for the result, you could use:

```
c->compute()
await c->compute_done

# at this point, c is done
# with its computation
```

After an `await`, `$agent`, `$name`, and `$value` correspond to the event that caused the `await` to finish. In the above example, `$agent` will be `c`, `$name` will be `"compute_done"`, and `$value` will be whatever value the `compute_done` event had.

Any other events that arrive during an `await` are still processed by Glish (i.e., it executes the body of any corresponding `whenever` statements). An `await only` statement can be used to tell Glish to drop these events instead. It is meant for use as a "hold-point", to freeze the effective execution of a Glish script until some seminal event occurs. Glish also provides a mechanism for listing exceptions to this rule, so that certain high-priority events will still be processed during an `await only`.

Point-to-Point Communication

Sometimes in a Glish system two clients need to communicate as fast as possible. If the system's Glish script only forwards events from one client to the other without modifying the events' values then we can instead use a direct connection between the two. Glish supports this style of communication using the `link` statement. When executed a `link` statement directs a client to send a particular event it generates directly to another client (perhaps renaming it). For example,

```
link t->transformed_data to
     d->new_data
```

will cause the client associated with `t` to send its `transformed_data` events directly to `d`'s client, which will see them as `new_data` events. (Other events generated by `t`'s client still go to the Glish interpreter.) The destination of a link can use the "`*`" event to mean "use the same name":

```
link t->transformed_data to d->*
```

will send the `transformed_data` events along without renaming them.

You can suspend point-to-point links with the `unlink` statement:

```
unlink t->transformed_data to
       d->new_data
```

suspends the link formed in the first example above. `t`'s agent will now instead send its `transformed_data` events to the Glish interpreter, which will execute the corresponding `whenever` bodies. Executing another `link` statement restores the point-to-point link.

Creating Agents

Agent values can be created three different ways. First, the predefined function `client` takes an *argv*-style list of strings and instantiates the corresponding program with the given arguments. `client` also has optional arguments for specifying on which host to run the process and whether to initially suspend the process to allow a debugger to be attached. For example,

```
t := client("timer", 5,
            host="psychosis")
```

runs the Glish client *timer* on the remote host "psychosis" with an argument of 5 (for "timer" this is the timer interval in seconds) and assigns to `t` an agent value corresponding to this process.

Another way to create an agent is to use the `shell` function with the optional argument `async=T` (`T` is the boolean "true" constant). Asynchronous shell clients can be sent `stdin` events to make text appear on their standard input, `EOF` events to close their standard input, and `terminate` events to terminate them. Each line of text they write to their standard output becomes a `stdout` event.

For example, here's a Glish script that uses *awk* to print the numbers from 1 to 32 in hexadecimal, each appearing as a separate event:

```
cvt := "awk '{printf(\"%x\n\",$1)}'"
hex := shell( cvt, async=T )

count := 1
hex->stdin(count)

whenever hex->stdout do
    {
    print count, "=", $value
    if ( count < 32 )
        {
        count := count + 1
        hex->stdin(count)
        }
    else
        hex->EOF()
    }
```

The first two statements associate an asynchronous shell client with the variable `hex`. The next line initializes the global `count` to 1 and sends that value to `hex`, making it appear on *awk*'s standard input.

The `whenever` body prints out the current count and its hexadecimal equivalent, and then either increments the count and sends *awk* a new input line or closes its standard input. Because Glish uses pseudo-ttys to communicate with asynchronous shell clients, *awk*'s output will be line-buffered, so each `stdin` event will shortly result in a new `stdout` event.

One might think that a race exists between sending the first `stdin` event to `hex`'s client and setting up the `whenever` to deal with the client's response. This problem does not arise, however, because the Glish interpreter does not read events generated by clients until it is done executing all of the statements in a script.

The final way to create an agent is using a *subsequence*. A subsequence is just like a function except that when called it returns an agent value, which can be used to send and receive events to and from the subsequence. In the body of a subsequence the predefined variable `self` refers to its agent value. For example, the sequence shown in Figure 4 creates two subsequences. When executed it prints: `36` followed by `[8 125 1030.3]`.

```
subsequence power(exponent)
 {
 whenever self->compute do
    self->ready( $value ^ exponent )
 }

square := power(2)
cube := power(3)

square->compute( 6 )
cube->compute( [2, 5, 10.1] )

whenever square->ready, cube->ready do
    print $value
```

Figure 4: Example of a Glish Subsequence

The first set of statements defines `power` as a subsequence that is invoked with an argument `exponent` and responds to `compute` events by generating a `ready` event whose value is the value of the `compute` event raised to the given exponent. The two assignments bind `square` and `cube` to agents corresponding to different instances of `power`. The next two statements send those agents `compute` events with a single integer value and a three-element double-precision array value, respectively. The final `whenever` statement prints the value of any `ready` events generated by `square` or `cube`.

5 The Client Library

Programs interface to the Glish system via the Glish "Client" library, which is written in C++. The library exports two classes: *Value* and *Client*. *Value* objects correspond with Glish values: they are dynamically typed arrays, records, functions, or agents. The *Client* class provides the mechanism for a Glish client to send and receive events.

The Value Class

Value objects can be constructed from C++ scalars or arrays. For example,

```
Value* v = new Value( 5 );
```

assigns to `v` a *Value* object representing the integer 5, while

```
double* x = new double[3];
x[0] = 1.0;
x[1] = 3.14;
x[2] = 4.56;
Value* v = new Value( x, 3 );
```

assigns to `v` the equivalent of the Glish value `[1, 3.14, 4.56]`. By default, *Value* objects constructed from arrays "take over" the array: they will *realloc* the array if it grows larger and delete it when the *Value* object is destroyed. The class library also provides mechanisms for specifying that an array should not be altered or should first be copied.

The *Value* class provides a number of member functions for manipulating values:

- `Type` returns the type of an object and `Length` its length.
- `IntVal` interprets one element of the value as a single integer, performing coercions as necessary, and similar functions are provided for boolean, floating-point, and string interpretations.
- `IntPtr` returns a pointer to a C++ array of integers that can then be used for direct access to the value's underlying elements, while `CoerceToIntArray` returns either the underlying array if already of type *integer* or else a copy of the array converted to *integer*. Again, these functions have counterparts for the other Glish types.
- `Polymorph` converts the value from its present type to a new type.
- Analogs to these functions are available for directly accessing and setting a record's fields.
- The function `create_record` (not a member function) returns a new, empty record.

A key point concerning the *Value* class is that it makes it easy to wrap Glish values around an existing program's data structures. These data structures can then be made available to other programs by sending them as event values.

Note also that both the *Value* and *Client* classes use reference-counting for memory management. The `Ref` and `Unref` functions manipulate each object's reference count. When the count reaches zero the object is deleted and any objects it refers to are `Unref`'d.

The Client Class

Each Glish client constructs one instance of the *Client* class by passing the *Client* constructor the program's `argc` and `argv`. When a Glish client is executed by a Glish script `argv` contains special arguments telling the *Client* object how to connect the Glish interpreter. So usually the beginning of a Glish client looks like:

```
int main( int argc, char** argv )
    {
    Client c( argc, argv );
    ...
```

The *Client* class provides three main member functions:

- `NextEvent` waits for the next event to arrive and returns its name and a corresponding *Value* object. The event is returned as a pointer to a *GlishEvent* object, which is simply a structure with `name` and `value` fields.
- `PostEvent` takes a string and a *Value* object and sends an event with the given name and value.
- `Unrecognized` is used to report that the current event is not recognized by the Glish client.

The class also provides variants on `PostEvent` for sending events with simple string values. In addition, the class provides access to the file descriptors from which it reads events, so the program can use *select* to multiplex between different input sources.

If the program was *not* invoked by the Glish interpreter then the special arguments will be missing. The *Client* library detects this case and knows that the program is running stand-alone, in which case it reads string-valued events from *stdin* and "posts" outbound events to *stdout*. This behavior allows client programs to be debugged separate from running within Glish.

6 An Example of a Client

Suppose we want to create an "FFT server": a Glish client that when sent a numerically-valued `fft` event computes the FFT of the array of data and returns the result as an `answer` event. The result consists of a record with two fields, `real` and `imag`, arrays of the real and imaginary parts of the Fourier transform.

Assume we have a function `fft` available for doing the actual transformation and want to "wrap" a Glish client interface around it. Figure 5 shows

how we would do so. First we create a *Client* object using the idiom discussed in the "The Client Class" section, earlier. We then enter the event-loop, blocking until a new event is ready (`NextEvent` returns a nil pointer when the client should terminate).

If the event's name is `fft` then we extract the event's value, convert it to "double" if it is not already, and extract its length into *num*. We then use `DoublePtr` to get a pointer to the actual array of double-precision elements. In order to call `fft` we need to also pass it arrays where it should put its results, so we create *real* and *imag*. After computing the FFT we create in *r* a Glish record value to hold the two arrays, and assign them to *r*'s `real` and `imag` fields. We then send this aggregate value as a Glish event with the name `answer`. Now that we're done with *r* we `Unref` it to reclaim its

```
#include <string.h>
#include "Glish/Client.h"

// Computes the FFT of the first "len" elements of "in", returning
// the real part in "real" and the imaginary part in "imag".
extern void fft( double* in, int len, double* real, double* imag);

int main( int argc, char** argv )
    {
    Client c( argc, argv );

    GlishEvent* e;
    while ( (e = c.NextEvent()) )
        {
        if ( ! strcmp( e->name, "fft" ) )
            { // an "fft" event
            Value* val = e->value;

            // Make sure the value's type is "double".
            val->Polymorph( TYPE_DOUBLE );
            int num = val->Length();

            // Get a pointer to the individual elements.
            double* elements = val->DoublePtr();

            // Create arrays for results.
            double* real = new double[num];
            double* imag = new double[num];

            // Compute the FFT.
            fft( elements, num, real, imag );

            // Create a record for returning the
            // two arrays.
            Value* r = create_record();
            r->SetField( "real", real, num );
            r->SetField( "imag", imag, num );

            c.PostEvent( "answer", r );
            Unref( r );
            }
        else
            c.Unrecognized();
        }
    return 0;
    }
```

Figure 5: Glish Wrapper for FFT Client

memory. This will automatically result in *real* and *imag*'s memory being reclaimed too. We don't need to `Unref` the *GlishEvent* pointed to by `e` because the next call to `NextEvent` automatically does so.

Finally, if the event wasn't `fft` then we inform the *Client* library that we don't recognize this particular event.

7 Implementation

The Glish language is implemented as an interpreter, written in about 10,000 lines of C++. It runs on SunOS, Ultrix, and HP/UX platforms; Glish clients can also run on VxWorks, using a limited client library written in C instead of C++.

We chose an interpreter implementation because it gives very fast turn-around times when modifying scripts, as well as the ability to run interactively. The interpreter is optimized to perform array-wise operations in tight loops, making its overhead acceptable.

In general inter-client communication goes through the interpreter; the design is *centralized*, much like the designs of Field [10] and HP SoftBench [2]. Remote communication occurs via TCP sockets, assuring reliable delivery of events, while local communication uses pipes for added performance. Point-to-point links enable faster but less flexible communication. We implemented them using named pipes for same-host communication and sockets for remote communication.

To create clients on a remote host the interpreter first remotely executes a daemon on that host to execute and control processes on its behalf, much like the SPC daemon used by HP SoftBench. All event communication with remote clients is still done directly between the client and the interpreter via a socket connection.

Event values are sent using a self-describing dataset format called *SDS*, similar to netCDF [13]. *SDS* handles padding, byte-swapping, and floating-point representation differences, so it can be used to efficiently transmit binary data between heterogeneous architectures (e.g., VAX and SPARC). The *SDS* layer is written in C (about 9,000 lines).

8 Performance

To give a feel for Glish's performance, on an unloaded Sparcstation 2 sending an empty event back and forth to a "ping" client on the same machine for 1,000 round trips takes an average of 6.5 real-time seconds, for an event rate of around 300 events per second. Sending 8KB-events takes an average of 10.3 real-time seconds, for a rate of about 200/sec and a data rate of 800 KB/sec. The CPU times (user + system) were about 60% of the real-time timings.

When making the same timings between a Sparcstation 2 running the Glish interpreter and a Sun IPC running the "ping" client connected via Ethernet, we found real-times (excluding startup overhead for running the remote daemon) averaging 7.3 seconds for empty events and 26.2 seconds for 8KB-events. These values correspond to about 275 empty events/sec, 75 8KB-events/sec, and a data rate of 300 KB/sec, about 25% of the raw Ethernet bandwidth.

Note that these rates correspond to the performance available when using point-to-point links. Forwarding events via the Glish interpreter halves the rates.

9 Related Work

It is widely held ([11, 3, 7, 2, 10]) that to build flexible distributed systems the individual programs in the system should have no knowledge of the inter-program connections (i.e., where their input comes from and where their output goes to). Extending this notion with self-describing data to form a "software bus" is discussed in [5] and [12], the former a "concept" paper and the latter a description of a proprietary system.

Many approaches to building distributed systems rely on special operating system support or writing client programs in specialized languages. For our purposes it was important that the system be portable between different Unix systems without kernel modifications, and that we be able to incorporate into the system existing programs written in C, C++, or FORTRAN.

Four systems that work with little operating system support and can integrate existing programs are Tcl [8], HP Softbench [2], Linda [3], and Field [10]. Tcl and Field limit interprocess communication to strings, making efficient communication of binary data problematic. Tcl also does not provide mechanisms for starting new processes. HP Softbench communicates data via the file system, requiring operating system support for network communication. (The related HP Encapsulator [4], though, provides a nice way to integrate existing programs into the system.)

The essence of all of these systems is enabling event "producers" and event "consumers" to find one another and communicate. Glish's main contribution is that it also provides a powerful language for manipulating both interprocess connections and the contents of the data passed between programs. In this respect Glish makes it easy to integrate programs with *different* interfaces; Glish provides the "glue" to bridge the differences between what one program generates and what another program expects.

10 Present Status

What we have described is the third generation of Glish (the first, very different implementation is discussed in [9]). We found that redesigning the language twice, while painful, greatly clarified and enriched it in result. Glish now has the flexibility to accommodate our simulation and control applications. Glish is used to control testing of superconducting magnets at the Superconducting Super Collider Laboratory (SSCL) and for analysis, simulation, and control of the Advanced Light Source accelerator at the Lawrence Berkeley Laboratory (LBL). Its use has been growing and we anticipate continued evolution of the system.

The current release of Glish is version 2.1. Source code can be retrieved via anonymous ftp to `ftp.ee.lbl.gov`. The current release includes all of the features described above except that client event values whose types are functions, agents, or references are not yet supported. The release includes the *SDS* layer used for communication between heterogeneous architectures. The *SDS* code is quite old and never made it out of prototype because of its success; we are rewriting it.

Glish is part of *ISTK* (Integrated Scientific Toolkit), a software package developed primarily by SSCL and LBL. *ISTK* includes a class library for creating objects that automatically change their value when Glish events are received, and send out new Glish events when their value are changed by other means (such as a user-interface). *ISTK* also includes a corresponding graphics library for building user-interfaces, making it simple to tie buttons to arbitrary multiprocess actions, or to automatically update displays when the data they reflect has been altered by another Glish client (or the Glish script itself). Along with these libraries *ISTK* also includes Glish-related applications such as a program for multiplexing string-valued Glish events and keyboard input into terminal-based programs, and a program for displaying Glish events as they are sent between clients. *ISTK* is not yet ready for general release, though interested parties may contact the second author for further information.

11 Future Work

A powerful addition to Glish would be having Glish clients "register" the events they respond to along with type signatures for those events, similar to the use of message patterns in Field, HP SoftBench, and Linda. Glish could then automatically connect together clients with similar event patterns, providing any necessary glue for accommodating differences. By including a "help" string with each registered event, the system could also interactively give the user help and type information on all events generated by all available clients. It then becomes possible to write a visual interface for composing Glish scripts, similar to that for Conic [6].

Other areas to explore are using shared memory for same-host communication (the *SDS* layer already supports this), out-of-band and prioritized events, richer exception handling than just `fail` events, and mechanisms for connecting multiple Glish interpreters together via events.

While we need more experience using Glish before finalizing the system design, our experiences to date have convinced us that with our software bus "shell" we have in place a firm foundation for building distributed applications.

12 Acknowledgements

We would like to thank our *ISTK* collaborators for their input and feedback on the design and use of Glish, especially Matt Fryer, Matthew Kane, and Mike Allen.

We would also like to thank Steve McCanne, Lindsay Schachinger, Matt Fryer, and the referees for their many helpful comments on various drafts of this paper.

For the curious, the Glish language was named by the second author so that when upper management types ask, "What language is this stuff written in?" we can reply, "*In Glish*, of course!"

References

[1] Richard A. Becker, John M. Chambers and Allan R. Wilks, "The New S Language", Wadsworth & Brooks, Pacific Grove, CA, 1988.

[2] Martin R. Cagan, *The HP SoftBench Environment: An Architecture for a New Generation of Software Tools*, Hewlett-Packard Journal, 41(3), pp. 36-47, June, 1990.

[3] Nicholas Carriero and David Gelernter, *Linda in Context*, Communications of the ACM, 32(4), pp. 444-458, April, 1989.

[4] Brian D. Fromme, *HP Encapsulator: Bridging the Generation Gap*, Hewlett-Packard Journal, 41(3), pp. 59-68, June, 1990.

[5] D. E. Hall, W. H. Greiman, W. F. Johnston, A. X. Merola, S. C. Loken and D. W. Robertson, "The Software Bus: A Vision for Scientific Software Development", Proceedings of the International Conference on Computing in High Energy Physics, Oxford, England, 1989.

[6] Jeff Kramer, Jeff Magee and Keng Ng, *Graphical Configuration Programming*, IEEE Computer, 22(10), pp. 53-65, October, 1989.

[7] Jeff Magee, Jeff Kramer and Morris Sloman, *Constructing Distributed Systems in Conic*, IEEE Transactions on Software Engineering, 15(6), pp. 663-675, June, 1989.

[8] John K. Ousterhout, "Tcl: An Embeddable Command Language", Proceedings of the 1990 USENIX Winter Conference, Washington, D.C., January, 1990.

[9] V. Paxson, C. Saltmarsh, M. Allen and M. Kane, *A Language, Server and C++ Class Library for Event Sequencing*, Nuclear Instruments and Methods in Physics Research, A293, pp. 356-362, 1990.

[10] Steven P. Reiss, *Connecting Tools Using Message Passing in the Field Environment*, IEEE Software, 7(4), pp. 57-66, July, 1990.

[11] Michael L. Scott, *Language Support for Loosely Coupled Distributed Programs*, IEEE Transactions on Software Engineering, 13(1), pp. 88-103, January, 1987.

[12] Dale Skeen, "An Information Bus Architecture for Large-Scale, Decision-Support Environments", Proceedings of the 1992 USENIX Winter Conference, San Francisco, CA, January, 1992.

[13] University Corporation for Atmospheric Research, "NetCDF User's Guide", available via anonymous `ftp`; retrieve pub/netcdf.tar.Z from host `unidata.ucar.edu`.

[14] Larry Wall and Randal Schwartz, "Programming Perl", O'Reilly & Associates, Sebastopol, CA, 1991.

Author Information

Vern Paxson holds an M.S. degree in computer science from U.C. Berkeley. He has been on the staff at the Lawrence Berkeley Laboratory since 1985, primarily working on software for accelerator physics simulation and control. Since 1991 he has also been a Ph.D. student at U.C.B. in the area of wide-area networking. Vern's claim to net-fame is as the author of *flex*, a high-performance *lex* rewrite. Reach him at `vern@ee.lbl.gov`.

Chris Saltmarsh obtained his doctorate at Nottingham University in cosmic ray physics, and then worked at CERN, firstly on the NA7 pion form factor experiment and then in the operations and machine physics groups of the Super Proton Synchrotron. He moved to the U.S. some 5 years ago to work with the Central Design Group of the Superconducting Super Collider and is presently with the SSCL working from the Lawrence Berkeley Laboratory. Reach him at `salty@largo.lbl.gov`.

Both authors can be reached via U.S. mail at Lawrence Berkeley Laboratory, MS 46-A/1123, 1 Cyclotron Rd., Berkeley, CA 94720.

Appendix: Glish Syntax and Grammar

The Glish syntax is free-form. Comments begin with '#' and extend to the end of the line. Statements are formally terminated with semi-colons but in general Glish is able to infer the end of a statement and supply an implicit terminator at the end of a line. Identifiers are case-sensitive; record field names and event names have separate name spaces and may include keywords.

In the following grammar, [ ]'s surround optional elements and {}'s surround elements that may occur zero or more times. Terminals are surrounded with quotes or appear in uppercase.

```
program:  { stmt }

stmt:  "{" { stmt } "}"
     | WHENEVER ev-list DO stmt ";"
     | LINK ev-list TO ev-list ";"
     | UNLINK ev-list TO ev-list ";"
     | AWAIT ev-list ";"
     | AWAIT ONLY ev-list
           [EXCEPT ev-list] ";"
     | event "(" [param-list] ")" ";"
     | IF "(" expr ")" stmt
                  [ELSE stmt]
     | FOR "(" ID IN expr ")" stmt
     | WHILE "(" expr ")" stmt
     | NEXT ";"
     | BREAK ";"
     | RETURN [expr] ";"
     | EXIT [expr] ";"
     | PRINT [param-list] ";"
     | LOCAL id-list ";"
     | expr ":=" expr ";"
     | expr ";"
     | ";"

expr:  "(" expr ")"
     | expr logop expr
     | expr relop expr
     | expr arithop expr
     | expr ":" expr
     | expr "[" expr "]"
     | expr "(" [param-list] ")"
     | expr "." FIELD-ID
     | unaryop expr
     | "[" "=" "]"
     | "[" [param-list] "]"
     | function
     | LASTEVENT
     | ID
     | CONSTANT

logop:  "|"  | "||"  | "&"  | "&&"
relop:  "=="  | "!="  | "<"  | "<="
      | ">"  | ">="
arithop:  "+"  | "-"  | "*"  | "/"
        | "%"  | "^"
unaryop:  "-"  | "+"  | "!"  | ref-type

function:  func-head "(" [formal-list] ")"
           func-body

func-head:  FUNCTION [ID]
          |  SUBSEQUENCE [ID]

func-body:  "{" { stmt } "}"
          |  expr

formal:  [ref-type] ID ["=" expr]
       |  "..."

ref-type:  VAL  |  REF  |  CONST

param:  expr
      |  ID "=" expr
      |  "..."

event:  expr "->" EVENT-ID
      |  expr "->" "[" expr "]"
      |  expr "->" "*"

ev-list:  event ["," ev-list]
id-list:  ID ["," id-list]
param-list:  param ["," param-list]
formal-list:  formal ["," formal-list]
```

UNIX Services for Multilevel Storage and Communications Over a Secure LAN

Bruno d'Ausbourg & Christel Calas – CERT-ONERA

ABSTRACT

In this paper we suggest a way to build very secure Unix systems and applications. They are based on the architecture of a highly secure machine (M2S) and the design of highly secure communicating mechanisms over a LAN architecture. The entire security is achieved and managed by a reduced Security SubSystem (SSS) operating in the hardware layer. This security is formally defined and is directed at the protection of both confidentiality and integrity of data, processes, and communication. The controls enforced by this SSS are founded on the rules of multilevel security.

The goal is to control causal dependencies inside the entire system by controlling all of the elementary flows of information. This leads to a machine and to a basic system for communication without any possible disclosure or unauthorized modification of information. In fact, no covert channel can be used in order to strike a blow at this system. Upon such a machine, services and functions of a Unix operating system can be built. And upon such a LAN, services and functions of a network system can also be built. We demonstrate what kind of services may be offered to the users and how they can be used to develop applications with new security features.

1 Introduction

Achieving storage and communications of data with a high degree of security is a need for more and more users, and becomes a goal for many developers since the Orange Book [DoD85] and then the Red Book [NCS87] have appeared. This need is currently answered by use of cryptographic features. But these techniques are quite inefficient when a high level of protection is required. Indeed, cryptographic techniques are useful for protecting the integrity or the confidentiality of any information which resides on a storage or communication medium. But such protection disappears when information is processed inside the system (computing or communicating system).

So, other approaches must be envisioned in order to ensure confidentiality and integrity inside computer systems. In particular, some of them attempt to build security functions and mechanisms inside the operating system itself. But in this case, these modifications are not always based on a good definition of the desired security, and the security controls enforced by the operating system can be bypassed, particularly by exploiting covert channels. This leads to illicit flows of information. Another approach consists of developing a hardware trusted computing base upon which an operating system must be built, integrating some light trusted parts. The Lock project described in [BLS89] and in [SW88] illustrates these new trends.

In this paper, we suggest a similar way in order to build very secure systems and applications. The latter is based on the architecture of a highly secure machine (M2S) and the design of highly secure communicating mechanisms over a LAN architecture. The entire security is achieved and managed by a reduced Security SubSystem (SSS) which operates in the hardware layer. This security is formally defined and aims at the protection of both confidentiality and integrity of data, processes, and communication. Controls enforced by the SSS are founded on the rules of multilevel security. The goal is to control causal dependencies inside the entire system by controlling all the elementary flows of information. This leads to a machine and to a basic system for communication without any possible disclosure or unauthorized modification of information. In fact, no covert channel can be used in order to strike a blow at this system. Upon such a machine, services and functions of an operating system can be built. Upon such a LAN, services and functions of a network system can also be built. We demonstrate what kind of services may be offered to the users and how they can be used to develop applications with new security features.

In Section 2, we explain the definition of multilevel security upon which the whole system is based. The desired security is founded on the control of user observation by enforcing control of causal dependencies. We describe these controls in the framework of a single machine, as M2S, and of a LAN. Section 3 details the architecture and the functioning of the entire LAN. In Section 4, we describe services offered by an operating system running on M2S: they permit multilevel storage for data and multilevel communications between hosts or processes which are classified at various security

levels. Section 5 discusses an application example, GDoM, using them.

2 Multilevel Security Definition

Security

The desired security protects both confidentiality and integrity of data and processes in the system. The formal definition given in [Eiz89] and [BCE90] establishes, with regard to confidentiality for example, that a system is secure if and only if the set of all the objects that may be observed in the system by a subject *s*, *O(s)*, is included in the set of objects he has the right to observe *R(s)*:

(1) $O(s) \subseteq R(s)$

It is important to constrain the definition of *O(s)* very closely. If not, a subject s can observe some object *o* not in *O(s)*. This object o can be used by a trap or a Trojan horse to disclose secret information.

Observation and Control of Causal Dependencies

A user is able to perceive the values of various objects anywhere in the system. Some of them may have a finer granularity than files. For example, in a workstation a user can observe the status of processes, of data structures such as a lock or a semaphore, of memory cells, or of registers inside a disk controller. He can also observe the duration of operations such as disk and memory accesses. On a LAN, he can observe collisions on a medium and the duration of elementary send and receive operations.

In fact, he can observe the value of data at various given times and perceive the times of their changes. Therefore, what may be really observed in the system is more than mere objects, but **points** *(object, time)*. Indeed, saying that a point *(o, t)* may be observed by a subject s involves two kinds of possible observations which entail two kinds of communication channels:

- the **value** of the object *o* at time *t* may be perceived, and a storage channel is involved here;
- the **time** or date *t* at which the object *o* takes a given value may be perceived, and a timing channel is involved here.

So, *O(s)* comprises points *(o, t)* that must be understood as values of objects *o* at a given time *t*. Output objects may be directly observed by a user and then their associated points *(o, t)* belong to *O(s)*. These points are produced by computations from other points reflecting the state of internal objects. These internal objects are themselves produced by computations from input data. So, *O(s)* contains more than points that can be directly observed: it also contains the points on which the points which may be directly observed are dependent. A precedence order on instants t defines as causal these dependencies in the model.

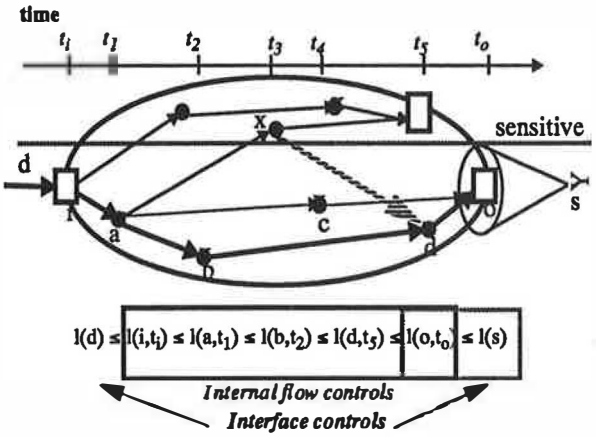


Figure 1: Causal dependencies inside a system

In order to maintain the set of objects that a subject s can observe *O(s)* in its rights *R(s)*, it is necessary to control these causal dependencies inside the system.

Protection by Levels

The use of levels ensures a good mastery of dependencies and of associated information flows inside the system. A security level (both in confidentiality and integrity) is attributed to objects and subjects. A subject with a level *l(s)t is allowed to observe only system points (p, t)* whose level *l(p, t)* satisfies:

(2) $l(p, t) \leq l(s)$

This inequality (2) must remain true for all points which are observable by the subject inside the system. Referring back to Figure 1, this condition leads to the conclusion that the inequality (3) must be satisfied inside the system:

(3) $l(d) \leq l(i,ti) \leq l(a,t1) \leq l(b,t2) \leq l(d,t5) \leq l(o,to) \leq l(s)$

In particular, it is forbidden for a point (d, t5) to causally depend on a point *(x, t3)* with $l(x, t3) \geq l(d, t5)$. Recall that this causal dependence could be:

- the value of object *d* at *t5* depends on *(x, t3)*;
- the value of time *t5* at which the object *d* takes a particular value depends on *(x, t3)*.

This would enforce a potential information flow from the sensitive point (x, t3) down to a not sensitive one *(d, t5)* and would be contrary to the definition of the security previously given. In fact, it follows that point *(o, to)* depends on no sensitive point in the system and its observation will reveal no sensitive information.

So, inequalities at system interfaces as described in Figure 1 can be enforced by classical techniques of physical and administrative security. The control of causal dependencies (temporal aspects included) is achieved by making sure every transition in the system. These transitions produce new points and make a transfer of information until system points directly observed by a user. All information channels are involved (storage and timing) and no potential covert channel exists. Values of levels

constitute a public (unclassified) information: the level of levels is unclassified.

Control of Information Flow

The control of information flow is a practical way to enforce the control of causal dependencies inside a system. Building a Security SubSystem (SSS) is a technical means to implement it. The SSS is in charge of managing the entire security inside the system. It drives all the mechanisms upon which the control of information flow relies. The next paragraphs explain principles of the SSS functions inside a multilevel machine and inside a multilevel communicating device which were developed at CERT/ONERA.

Inside a Machine for Multilevel Security: M2S

M2S discussed in [AL92] combines a processor with an address space A. The processor addresses the space A when executing elementary transfers to external devices such as memory or registers of controllers.

The elementary objects that can be observed by a user comprise processor registers and cells of the address space. Levels are assigned to these objects. The level assigned to the processor registers determines the *current level cl* of the whole system. A value is assigned to cl by SSS in accordance with rules of time multiplexing. Assigning levels to cells of the address space divides it into different partitions. Each partition of this address space may be reached by the processor according to the value of its current level *cl*, the requested access mode, and rules of flow control.

The state of the system is reflected by the state of processor registers and the state of buses (address, data, and control). Inside the system, internal information flows are caused by elementary transfers between the processor and the address space: transfers of data or interrupt signals. The SSS is in charge of controlling these information flows. It does this by making use of specific hardware components which are under control of a Security Processor (PS). This PS uses its own resources in order to store and manage security data.

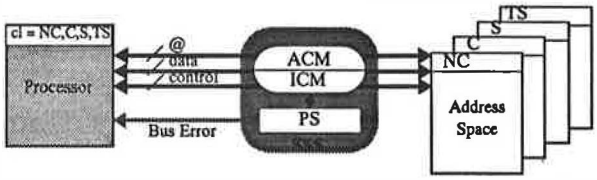


Figure 2: Elementary transfer controls inside the system

Figure 2 describes how the SSS controls the information flows inside the system. In fact, the SSS inspects the state of buses in real time. It determines which states are allowed in accordance with security data and with rules of flow control. If an illicit state is reached during an elementary cycle, the cycle is interrupted by the PS and a Bus Error signal is sent to the Processor.

The Access Control Module (ACM) controls the first kind of transfer inside the system. At current level cl of the processor, a read (or write) cycle to an address *na* will be allowed if the following conditions(4) (or (5)) are satisfied:

(4) $cl \geq na$

(5) $cl \leq na$

This module comprises an additional component which is in charge of controlling transfer operations that use a more complex addressing mode. In particular, when an access to a disk data block is involved, the processor uses the data bus in order to transfer some addressing information to the disk controller, e.g., cylinder, track number, or sector number.

The Interrupt Control Module (ICM) controls the second kind of transfer. It filters interrupt signals emitted by peripheral devices which are located somewhere in the address space. If the signal sender is an object whose level is lo, the interrupt signal is also an object whose assigned level is $li = lo$. ICM transmits this signal to the processor when its current level *cl* satisfies:

(6) $cl \geq li$

In any other case, the signal is suspended until condition (6) becomes valid. In practice, in order to handle them more easily, interrupt signals are received by the processor when:

(7) $cl = li$

Inside a Communicating Device

The programming model in use combines a communicating device with a communication bus addressed when executing elementary exchanges with other devices over the LAN. The medium access control method envisioned is CSMA/CD.

The communication device is an active entity and acts as an agent for the elementary communication system. Objects that can be observed are composed of the communication bus and the registers and memory cells of all of the devices. These memory cells have messages to send or receive stored in them. Object values at various times make up the different system points. Lastly, levels are assigned to devices and the bus. Levels are themselves security objects. The device level determines its current level.

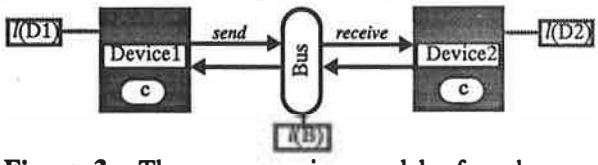


Figure 3: The programming model of exchanges over the LAN

Two operations are defined over such a subsystem and are exerted by communication devices: they can *send* or *receive* a message.

Sending Operation

The activation of this operation at *to* entails a modification of the object B(us) by an agent D(evice). D deposits a message m on B which was previously stored in a memory cell C of D.

(8) $l(D, to\text{-}1) = l(B, to\text{-}1)$

In other words, a communication device may send at to any message only if the communication bus is at its own level.

Receiving Operation

At *to*, receiving any message by a communication device D consists of observing the state of the bus B and collecting bits circulating on it and registering them in cells of D.

(9) $l(B, to\text{-}1) = l(B, to) \leq l(D, to\text{-}1)$

In other words, the only devices allowed to listen the bus are those whose level dominates the level of Bus. That's a good natural condition.

3 System Overview

General Architecture of a Secure Multilevel LAN

Figure 4 describes the general architecture of a secure multilevel LAN. The entire security is based upon the existence and the functioning of a Distributed SSS (DSSS). Such a LAN is composed of standard workstations which are variously classified. These hosts are not trusted, and neither are their network interfaces. The LAN may comprise also a secure multilevel host acting as a multilevel server, for example. We detail in [AL92] the architecture and the functioning of a multilevel station: M2S.

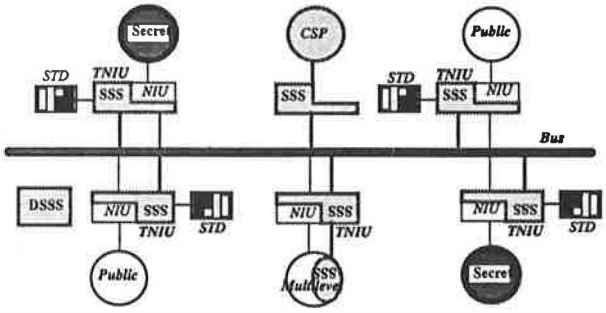


Figure 4: Structure of a secure multilevel LAN

The trusted network interface unit (TNIU) is the basic component in this architecture. Such an interface device integrates the host network interface unit (NIU) and a local security subsystem (SSS) whose job consists of upholding security conditions for exchanges carried on by the communication medium. So, this local SSS controls accesses to the medium exerted by the interface unit in such a way that they stay in accordance with information flow control rules which it is charged to enforce. It inhibits or frees physical access to the bus in order to control the receipt or transmission of elementary operations. Its activation is engaged on the basis of security data related to level values assigned to the interface unit (and the host) and the bus. These values are established, managed and provided by the CSP by mean of exchanges of security data. Transfers to and exchanges with CSP are regulated by a specific communication protocol we call SMAC (Secure MAC).

Such an interface unit is coupled with an annex security token device (STD) whose function consists of providing a secure path to the DSSS for the host user. In particular, as it is directly connected to the local SSS, it is able to realize operations such as authenticating users at the hosts and securely reserving a session level for host communications.

The local SSS relays the security policy established and enforced inside a Central Security Processor (CSP) and executes instructions it receives from the CSP.

One way to achieve the integrity on security data exchanges consists of isolating this security communication channel from the user communication channel. A logical separation is enforced, based on a mode of use for the bus which is time multiplexed by CSP and which may temporarily be reserved for exchanges by security subsystems.

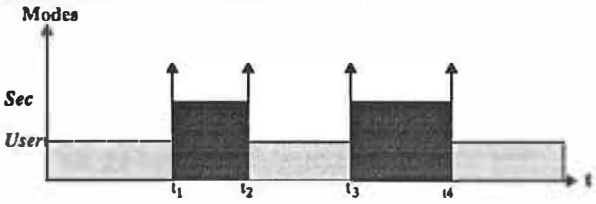


Figure 5: Multiplexing modes for the bus use

Global Function Description

At the initial state, the LAN bus is at a public level. When initialized, a local SSS inhibits send and receive operations for the interface unit until valid information related to the bus level enables it to configure the interface unit in accordance with multilevel security rules.

When a user on an host, classified or not, requests access to the LAN, he must use the STD in order to send a session reservation for the chosen level *ls* to the CSP. The STD uses the SMAC protocol interface in order to emit its public request to the local SSS which transmits it to the DSSS. This session level *ls* must be dominated by the classification level attributed to the host and may be the minimal one, public, or a classified one. In the second case, the reservation data must contain any information about the duration for the requested session. The CSP registers this request and plans the entire temporal multiplexing for the bus resource.

If *ls* is the actual current level of the bus, the CSP may transmit to the local SSS a functioning

permission for a given time. This time computation must take into account the time yet passed at this current level and the time the message will be received by the local SSS in order to maintain the consistency of the temporal multiplexing policy enforced by the CSP.

When *ls* is not the current level, the local SSS must wait for a message issued by the CSP to the all local SSSs to announce to them a new current functioning level *(cl, t)* for a given time *t*. Local SSSs react to a such information by inhibiting access to the bus for sending or receiving and by positioning the network interface unit according to the levels of the bus and the current session. An impulse on the Bus issued by the CSP acts as a clock tick and starts the current session.

At the end of time t, the local SSS stops the current session and enforces a security functioning mode by inhibiting send operations. Then the SSS waits for new instructions from the CSP. The timing control is enforced by the local SSS. In this way, in any case of a problem in the CSP or on the bus, the interface unit will not continue emitting at session level ls on a bus whose current level cl would have been changed to a new value $cl \leq ls$.

The new session declared by the CSP may be a public level or not. If it is a public level, it must be declared for a given time *t*. If $t=0$, that means a return to the initial state and the temporal multiplexing is relaxed. If $t \neq 0$, temporal multiplexing is currently running and this session will be chained to another one at the end of the time limit.

In all cases, the local SSSs maintain the access configuration of the interface units they control in accordance with level information in a security functioning mode. An impulse issued by the CSP causes this functioning mode to change into a user one.

In case of an error or bad CSP frame receipt, the local SSS blocks the interface unit and inhibits any receive or send operation over the bus. It then tries to warn the CSP by issuing it an error signaling frame.

So, a multilevel station, such as M2S, is able to communicate with various stations at different levels. In each case, a communicating process is involved on the station at the same level as the bus resource and the peer station. The control of information flows is enforced inside this station and inside the communicating device by the respective SSSs. Then, such a machine may be used securely in order to offer multilevel file system services, for example.

4 Unix Multilevel Services

A Unix operating system was developed on this machine. It offers both classical Unix services and multilevel security services which allow one to take advantage of the possibilities of M2S and the LAN architecture.

Storage of Multilevel Data

We expect a multilevel system to furnish storage for differently classified data. We propose two kinds of Unix structures inside which these data can be collected. The former is a classified regular file and the latter is a single entity, called a multilevel file, which can contain variously classified information. Both kinds of files and classified directories constitute the Unix multilevel hierarchy which ensures a strict partitioning of data according to their classification level.

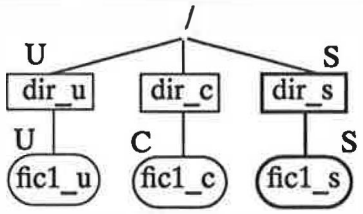


Figure 6: UNIX multilevel hierarchy example

Classified Regular Files

Classified regular files can be created inside a directory with the same level of classification. Basic objects used to construct a classified file or a classified directory are: classified disk blocks, a classified inode, and a public reservation inode. The latter is used to declare and introduce the existence of a secret directory at a public level. Indeed, in order to create a secret directory, a public process has to find a free secret inode capable of assuming this directory. However, being at a public level, it cannot read the secret inode table and so it cannot find a free one. We introduce public reservation inodes in order to resolve this problem. They are images of secret inodes. The public process examines them and when it finds a free one it knows that the corresponding secret inode is free too. Hence, knowing a free secret inode it can realize its initialization blindly. The real creation will be done later by a secret process [Aus92].

Users manage the Unix hierarchy through special services. These are:

smkdir(dir_name, level)

which creates a classified directory from a public level, and *P and*

creat(), unlink(), open(), read(), write(), seek() close()

which have the same syntax as the classical calls but have different semantics.

Assume we have the Unix hierarchy presented in Figure 6. Realization of previous services depends on the current level of their execution and level of the target file or directory. According to security rules presented in 2, users can read lower files or

lower directory contents and modify files or directory contents at the same level. Thus, at current level C, we obtain the following results from the presented system calls:

open("/dir_u/fic1_u", "r")
returns classical file descriptor.

open("/dir_u/fic1_u", "w")
returns "Write not allowed at current level".

open("/dir_c/fic1_c", "rw")
returns classical file descriptor.

open("/dir_s/fic1_s", "r")
returns "File unknown".

creat("/dir_u/fic2", ...)
returns "Creation not allowed at current level".

creat("/dir_c/fic2_c", ...)
returns classical file descriptor.

These controls rely upon those enforced by the SSS when the kernel tries to access basic objects which compose files and directories *(disk block, inode in block memory, etc.)*. So the security is not enforced by the Unix kernel, which therefore can be totally untrusted, but by the SSS only.

Multilevel File

The purpose of the *multilevel file* is to collect different level information inside a special Unix file, thus providing complete multilevel security. In particular, such a structure would be of interest in a multilevel mail system to store messages or documents which contain unclassified information like sender identity, receiver address, and variously classified texts. Such a message could be stored inside a single Unix secret file but it would not be satisfactory since unclassified parts *(overclassified in secret data)* could not be reached by unclassified processes *(a mail server for instance)*.

Multilevel files are a single entity composed of classified areas called sections (see Figure 7). There is a single classified section by level containing all the file information classified at this level. The SSS ensures multilevel file security by managing access to its sections. In order to obtain this simple function, we implement the Unix multilevel file structure respecting some rules described below.

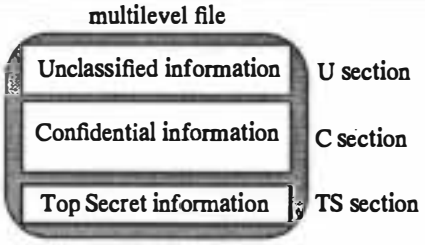


Figure 7: Multilevel file organization.

We use the same Unix structures as those which are used for single classified files: namely classified disk blocks, and classified and reservation inodes on which we made some little modifications. These modifications were realized in such a way that they maintain complete compatibility with classical Unix structures.

M2S offers multilevel systems of files whose access is controlled by the SSS. Every section of a Unix multilevel file is stored in a subtree of the whole file system; this subtree is classified at the level of the section. The SSS does not distinguish file blocks and section blocks so that it enforces the security of monolevel and multilevel file blocks without any change (see Figure 8).

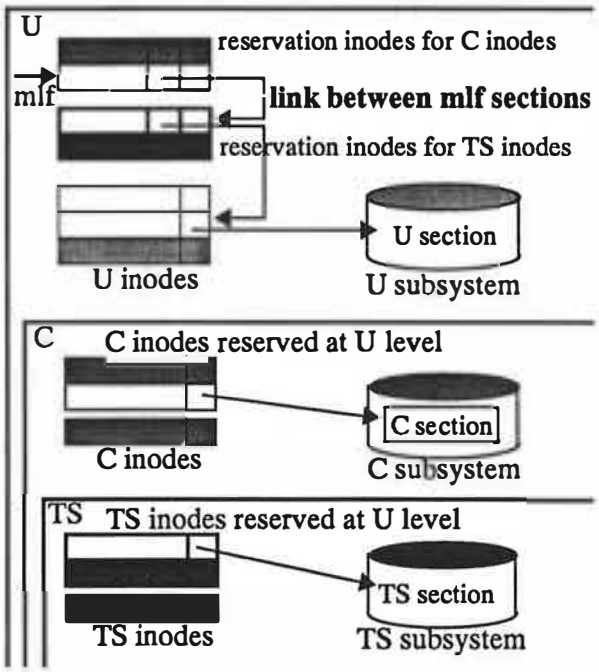


Figure 8: Multilevel file implementation example

Furthermore, for each section, we use an inode classified at the section level which maintains the block disk list, section characteristics *(size, owner, etc.)*, and a public reservation inode. The use of public reservation inodes is unavoidable since the Unix kernel must maintain a link between the inodes of a given multilevel file in order to maintain which inode manages what section. This link can be maintained only between inodes with the same classification and therefore we use reservation ones which are both at the same public level and permitted to manage higher blocks. Figure 8 shows an example of the implementation of a multilevel file "mlf" containing U, C, and TS sections.

Some internal services were also modified and adapted like *namei()* adding one parameter which indicates the section level wanted for a given multilevel file. New internal services like *iralloc(), ilink() and iunlink()* were introduced which permit the creation, linking, and unlinking, respectively, of public reservation inodes.

From previous presentation it follows that section creation and destruction have to be done at a public level since they need the modification of a public reservation inode. Thus a multilevel file could be created only in a public directory. To manage the multilevel files, users have special primitives and commands:

screat(mlf_name, rights, section_level)
creates a section inside a multilevel file.

sunlink(mlf_name, section_level)
deletes a section inside a multilevel file.

sopen(mlf_name, mode, section_level)
opens a section returning a descriptor.

sexist(mlf_name, section_level)
tests if a section exists or not in a multilevel file.

read(), write(), seek(), close()
use section information through a descriptor.

mkfmn mlf_name section_level
creates a section inside a multilevel file.

rmfmn mlf_name section_level
removes a section inside a multilevel file.

stfmn mlf_name
shows the section list.

Classical Unix commands yet work on a multilevel file but their behavior has been modified. For instance, the cat command applied on a multilevel file displays the information of all sections dominated by the execution level of the command.

Multilevel Processing

The second functionality that a multilevel system must offer is support of concurrent multilevel processing. These processes can be created by the system itself at user login time or dynamically created by another process. This creation can be made through two primitives: *fork()* and *sfork()*.

The former has a classical syntax and permits creation of any child process at the same level whereas the latter allows a public process to create secret ones. Its syntax is *sfork(child_level, child_duration)*
where *child_level* indicates the level needed for the child and *child_duration* the quantum of secret time that the father reserves for its child. This one being at higher level than its father it cannot indeed signal the parent when it dies. So after the duration has elapsed, the father can consider that its child is dead. The secret process creation uses multiplexed structures and reservation in advance of secret resources *(time quantum, memory, etc.)* from the public level. A closer description can be found in [AL92] [Aus92].

Multilevel Communication

The third multilevel functionality is the ability to perform communication between remote processes across the special multilevel network presented in 3. This network provides special characteristics at the lower layers of communication *(physical, MAC)* and we are concerned here with the repercussion of this behavior at the transport layer.

A multilevel network permits the exchanges between two hosts running at the same level; so the transport layer offers only communication between two processes classified at the same level. Particularly, we deal with Unix socket structures and services and we explain how we take account of this special functioning over the network.

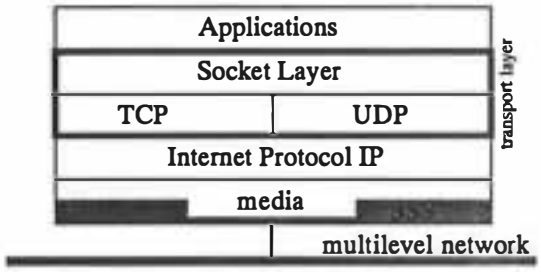


Figure 9: Unix-Internet decomposition

Multilevel Socket

In UNIX, any process desiring to communicate with another remote one must create a memory structure called a TSAP (*Transport Service Access Point*). We are concerned in this paper only with socket TSAP.

Any Unix remote communication is realized from a socket to another socket. Thus when a process needs to send data to another process, it must send the data to a socket belonging to the target process. A TSAP is bound with an external address. So this address must be used by any remote process wanting to reach the owner process of the socket bound to this address. There are two kinds of external addresses which differ by the sort of processes reachable through them. On one hand is the Unix address type *(special file name)* which is used to communicate with processes running on the same host, and on the other hand the Internet address type *(port number)* used to communicate with processes on remote hosts.

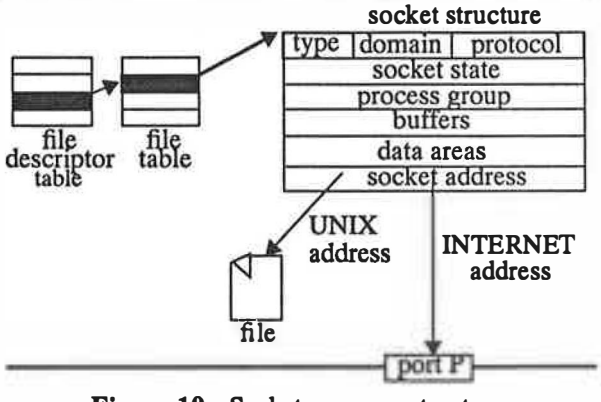


Figure 10: Socket memory structure

The socket memory structure presented in Figure 10, including buffers and data areas, is stored in memory blocks classified at the same level as the owner process's clearance. The external address is also classified. A classified Unix file is used in a Unix address and the port number of an Internet address is multiplexed by level. Therefore the same port number can designate several processes according to the value of the current level (see Figure 11).

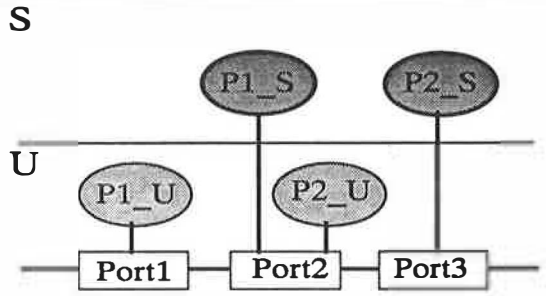


Figure 11: Example of a multiplexed port number

Remember that the multilevel network allows communication only when the two hosts run at the same current level. Consequently, no matter what port number of the address is indicated, a process can only reach another process if it has the same classification. For example, in Figure 11, a remote process S sending data to the socket *port2* reaches *P1_S* and never *P2_U*, which stays out of reach.

Multilevel Socket Services

Concerning the socket primitives, they all keep the same syntax but some of them take different semantics. First of all, a process must create a socket. For that it uses:

socket_desc = socket(domain, type, protocol)

which returns the file descriptor used to designate the created socket. An error can be returned in case of inadequate memory space for the structure. The process must then bind to this socket with an external address in order to allow its access from any remote hosts. The call

bind(socket_desc, address, lg_address)

realizes this binding. For Unix addresses the file name specified must be classified at the process level; otherwise it returns an error. For Internet addresses, it can indicate any port number. All are valid since they are multiplexed by level. However, the kernel can return an error in case a port number is already in use by another socket. The communication may take the two standard forms: unconnected mode or connected mode. To realize these communications, processes use the following primitives:

sendto (socket_desc, message, lgmsg, options, receiver_address, lg_address)
recvfrom (socket_desc, message_buffer, lgbuf, options, sender_address, lg_address)

to exchange data in unconnected mode,

listen (socket_desc, connection_number)
socket2_desc = accept(socket_desc, requester_address, lg_address)

to initialize the server,

connect(socket_desc, server_address, lg_address)

to request the connection with a server,

send(socket_desc, message, lgmsg, option)
recv(socket_desc, message_buffer, lgbuf, option)

to exchange the data in connected mode, and finally

shutdown(socket_desc, allowed_operation)

to close the connection.

All these services and any others not presented here keep the same syntax as the classical ones. However a behavior change appears for all the ones which need external address parameters. The same behavior is kept with Internet addresses but they return an error when the Unix file name of a Unix address is not classified at the same level as the process clearance.

5 Application Example

UNIX Services

Each machine must be able to offer services to other machines running any process which requests it. To realize that, a special process is executed on the machine which runs as a server waiting for requests. This is only possible when the remote processes know the port number on which this server waits.

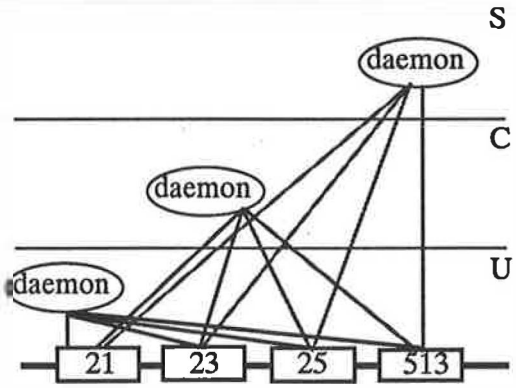


Figure 12: Classified daemons example

In Unix, a normalization defines that some port numbers are reserved for special services and so cannot be used for user sockets. The port numbers reserved are fixed at boot time between 0 and IPPORT_RESERVED. The UNIX normalization defines some services that constitute Internet standard services and Unix services. Amongst them we find FTP *(file transfer protocol)*[port 21], SMTP *(simple mail transfer protocol)* [25], TELNET [23] and NFS *(network file system)*. In a multilevel system, we saw that port numbers are multiplexed by level including reserved ports between 0 and

IPPORT_RESERVED. To provide a service at a given level, a server must be created at this level.

Unix uses a special server, called the *inetd* daemon, which awaits connection on several port numbers concurrently. It creates the server for the corresponding port *(whod, for instance, if the request arrives on port 513)* and puts itself back into a wait state again. As presented in Figure 12, the multilevel system uses also this principle. It initializes as many inetd daemons as there are different levels in the system, each of them creating servers at its own level, by which classified Unix services may be obtained.

GDoM Example

We discuss then a concrete application which runs above such a system, realizing its work through the multilevel services exposed in previous sections. This application is GDoM *(Gestionnaire de Document Multiniveau)*, a multilevel document manager [Cal92] specified and developed at CERT-ONERA. We present some of its functionalities and describe briefly what kind of multilevel services are used to enforce them and how they are used.

GDoM permits the storage of differently classified information inside a library structure. This information is basically stored in pages. The pages of a given classification are collected inside a document and documents of various classifications are stored in a multilevel folder. The various cleared users can store or delete new information, read or modify them, and can transfer a folder from one library to another remote one. Each of these actions is controlled in order to maintain information security and to avoid any illicit information flow. In fact, these controls rely upon basic controls performed by the SSS. Hence, we obtain a multilevel secure behavior with an application which is absolutely untrusted.

Consider now the implementation of the GDoM entities, and how the actions presented previously are realized through the use of multilevel services, namely multilevel file services, multilevel processing, and multilevel communication.

Each folder is implemented by a multilevel file. To simplify, consider that there is only one document of one classification in a folder and a single page in the document. So a folder is implemented by a multilevel file, a document by a section of this multilevel file, and the page information is stored in this section. In reality, the folder is also a multilevel file but we have several documents of the same classification and several pages in each document. In our simplified framework, the creation of a document is realized by calling a multilevel service which was previously defined; see Figure 13.

Notice the use of the screat service which creates a section in a multilevel file. *Folder_name* is directly used as multilevel file name and *document_level* as the level of the section. This function will succeed only if it is called at a public level since it realizes a section creation. The destruction of a document is made by calling the sequence shown in Figure 14.

```
int creat_document (folder_name, document_level)
char *folder_name;
level document_level;
{
    /*--- section creation ---*/
    if (screat(folder_name, 0666, document_level)!=OK)
        return(ERR_CREAT);
    else
        return(OK);
}
```

Figure 13: Simple document creation

```
int destroy_document (folder_name, document_level)
char *folder_name;
level document_level;
{
    /*--- section destruction---*/
    if (sunlink(folder_name, document_level)!=OK)
        return(ERR_DESTROY);
    else    return(OK);
}
```

Figure 14: Destruction of a document

To read or modify a page, GDoM creates a new process which is in charge of doing the modification. This new process has the same classification as the GDoM process if it is a secret one (not unclassified) or has the classification of the page if GDoM is a public process. The creation of the new process is realized with the code in Figure 15. Figure 16 shows how to read the page information, we open the corresponding section and put its contents into a buffer. This function realizes the read operation only if the user calls it at a current level dominating the classification of the page. The storage of the modifications of the page information can be done with the code shown in Figure 18.

The transfer of a folder needs the transfer of a multilevel file through multilevel communication services. We do not present the source of the function here but we prefer to present its principles in Figure 17.

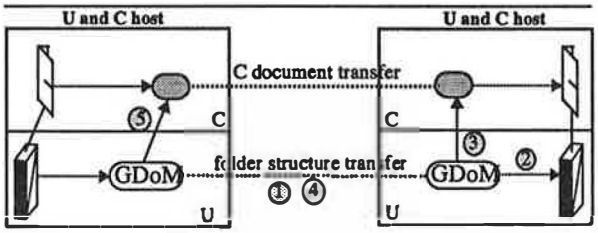


Figure 17: Principles of the transfer of a folder

```
int edit_page (folder_name, page_level)
char *folder_name;
level page_level;
{
    level start_niv;

    /*--- edition level setting ---*/
    if (get_current_level()!=PUBLIC)
        start_niv = get_current_level();
    else    start_niv = page_level;

    /*--- editor program launching ---*/
    switch (sfork(start_niv, EDIT_DURATION))
    {
    case -1 : return(ERR_EDIT);
    case 0 : execv("GDoMEditor", arg_set(folder_name, page_level));
    default : return(OK);
    }
}
```

Figure 15: New process creation

```
int get_page(folder_name, page_level, buffer)
char *folder_name;
level page_level;
char *buffer;
{
    int section_desc;

    /*--- section read opening ---*/
    if ((section_desc=sopen(folder_name, O_RDONLY, page_level))<0)
        return(ERR_READ);
    else    {
        /*--- get page content in the buffer ---*/
        read(section_desc, buffer, LG_BUFFER);

        /*--- close the section ---*/
        close(section_desc);

        return(OK);
        }
}
```

Figure 16: Section opening and reading

The transfer protocol can be decomposed in five phases:

1) The two public GDoM communicate. The sender sends to the target GDoM the composition of the folder *(list of its sections)*.
2) The target GDoM creates a folder with same sections as the source folder.
3) It creates as many servers as there are sections in the folder. Since they're at different levels, each server can wait on the same port.
4) The two GDoM exchange the contents of the public section.
5) The sender creates client processes in charge of contacting the servers and exchanging their sections.

6 Conclusion

We described in this paper how new multilevel services can be built on a secure system. This security is founded on a theory and uses levels in order to ensure both confidentiality and integrity of data and processes. When implemented, it relies upon a few control mechanisms which are driven by a reduced Security SubSystem. Such mechanisms may be located in the hardware layer because of the poor semantics of multilevel rules. So it becomes possible to obtain a very high degree of protection over a machine or a LAN: no illicit disclosure or unauthorized modification of information.

This high degree of protection does not entail any increase in the size or the development cost of operating systems and applications. On the contrary, it is possible to offer standard mechanisms, functions, and services of an operating system like Unix. In this case, since all the security is ensured by the SSS, there is no need for a trusted mechanism in the operating system. The latter must only be adapted to the function of the new virtual secure machine. So, these systems are able to run any existing Unix application. But the multilevel security enforced by the system also offers new services through the definition of new data structures and new functions. We illustrate this point by discussing multilevel files and multilevel sockets. These services permit the construction of highly secure applications which are able to answer some of the needs of users requiring good protection of their data, processes, and communication.

In fact, a new trend appears which states that some basic functions and mechanisms ensuring a very strong multilevel security may be integrated into the hardware architecture of computer systems, machines, or communicating devices. This integration leads to new architectures of processors. Then the problem becomes one of building Unix operating systems and Unix applications which take account of these new chips and their functioning. In fact, the problem is not yet in building secure Unix operating systems and applications, but in building Unix operating systems and applications on really secure hardware.

7 References

[AL92] B. d'Ausbourg, J-H. Llareus, "M2S : A Machine for Multilevel Security"; Proceeding of Esorics'92, Toulouse, November 23-25, 1992.

[Aus92] B. d'Ausbourg, "Unix Operating Services on a Multilevel Secure Machine"; Proceedings of USENIX UNIX Security Symposium, 1992.

```
int put_page(folder_name, page_level, buffer)
char *folder_name;
level page_level;
char *buffer;
{
    int section_desc;

    /*--- section write opening ---*/
    if ((section_desc=sopen(folder_name, O_WRONLY, page_level))<0)
        return(ERR_WRITE);
    else    {
        /*--- put buffer on page content ---*/
        write(section_desc, buffer, length(buffer));

        /*--- close the section ---*/
        close(section_desc);

        return(OK);
        }
}
```

Figure 18: Storing modified pages

[BCE90] P. Bieber, F. Cuppens and G. Eizenberg, "Fondements theoriques de la Securite Informatique"; Rapport 2/3366.00/ DERI, Centre d'Etudes et de Recherches de Toulouse, 1990.

[BLS89] J. M. Beckman, J.R. Leaman and O.S. Saydjari, "LOCK trak : Navigating Uncharted Space", IEEE Symposium on Security and Privacy, Oakland, 1989.

[Eiz89] G. Eizenberg, "Mandatory Policy: Secure System Model"; In AFCET, editor, European Workshop on Computer Security, Paris, 1989.

[Cal91] C. Calas, "Definition de mecanismes necessaires a la realisation d'un service de messagerie sur une machine Unix smn"; Rapport de DEA, Universite Paul Sabatier, Toulouse III, Juin 1991.

[Cal92] C. Calas, "GDoM, a Multilevel Document Manager"; Proceeding of Esorics'92, Toulouse, November 23-25, 1992.

[DoD85] Trusted Computer Systems Evaluation Criteria. Technical report DoD 5200.28-STD, National Computer Security Center, Fort Meade, MD, December 1985.

[NCS87] National Computer Security Center, "Trusted Network Interpretation of the Trusted Computer System Evaluation Criteria", NCSC-TG-005, July 1987.

[SW88] M. Schaffer and G. Walsh, "LOCK/ix : On Implementing Unix on the LOCK TCB", 11th NCSC Conference, 1988.

Author Information

Christel Calas is a Ph.D. Student at the National School for Aeronautics and Space in France. He is a member of the CERT/ONERA team in Computer Security. His current research interests are operating systems and distributed systems. He can be reached at CERT/ONERA, Departement d'Etudes et de Recherches en Informatique, 2 Avenue E. Belin, B.P. 4025, 31055 Toulouse, France, or electronically at calas@tls-cs.cert.fr.

Bruno d'Ausbourg is a Research Engineer at CERT/ONERA in France. He is a member of the Computer Security Team at CERT. His current research interests are operating systems, distributed systems, and designing and verifying network protocols. He can be reached at CERT/ONERA, Departement d'Etudes et de Recherches en Informatique, 2 Avenue E. Belin, B.P. 4025, 31055 Toulouse, France, or reach him electronically at ausbourg@tls-cs.cert.fr.

A Sketch Of The Smart Frame Buffer

Joel McCormack & Bob McNamara – Digital Equipment Corporation

ABSTRACT

Using a RISC processor to drive a simple frame buffer yields good 2D color graphics performance. But processor, memory, and bus architectures can prevent processors from saturating video RAM bandwidth. The smart frame buffer is a small cheap gate array that makes full memory bandwidth available to the CPU by expanding 32 data bits into operations upon 32 pixels; pixels can be 8, 16, or 32 bits deep. We avoid the cost and complexity of typical graphics accelerators by leaving high-level control to the CPU, yet achieve comparable performance. This paper describes the architecture of the smart frame buffer chip, sketches several software algorithms for common X11 graphics operations, and compares performance against other popular graphics hardware.

1. Introduction

In a previous paper [4], one of us described the dumb frame buffer hardware and software used on early Digital RISC workstations. This simple approach yielded cheap graphics with good performance, but the future of dumb frame buffers is not bright. Many new processors implement byte and other partial-word writes using painfully slow read/modify/writes, and even fast I/O busses provide only a small fraction of the bandwidth available from video RAMs in typical configurations. To fully exploit current VRAM technology under these constraints requires specialized graphics hardware.

The smart frame buffer is a small cheap gate array that locally expands 32 data bits into operations upon 32 pixels; pixels can be 8, 16 or 32 bits deep. This expansion enables us to provide information for 250 megapixels/second via the TURBOchannel bus. Different modes of operation provide support for filling solid areas, stippling areas, copying areas, and drawing solid and dashed lines. Complex operations, such as computing the shape of an object and the pattern to paint within it, are left to the CPU.

Limiting graphics assistance to a few simple commands reduces design time, increases reliability, reduces chip cost, allows designers to focus upon making VRAM bandwidth available to the CPU, and allows graphics performance to improve in tandem with CPU performance.

The smart frame buffer design proved all of these advantages. Initial design to power-up took 9 months. The chip contained two bugs, easily bypassed in software, and then fixed on the second pass. Chip cost is less than the external glue logic it obviates. The full video RAM bandwidth is available to the CPU for most operations. And performance of the sfb has steadily improved from the 25 MHz MIPS-based DECstation 5000/200 to the 40 MHz DECstation 5000/240 to the 150 MHz Alpha-based Flamingo. In fact, Flamingo performance exceeds all competitors on a majority of the graphics benchmarks included in the `x11perf` performance measuring tool. The smart frame buffer sets an aggressive new level of performance for "low-end graphics," and belies the common wisdom that graphics systems need to be complex to be competitive.

This paper describes the architecture of the smart frame buffer chip, sketches software strategies for graphics operations common in the X Window System, and compares performance against other popular graphics hardware. Finally, we summarize the reasons why such simple hardware performs so well.

2. Design Goals and Strategies

In priority order, our design goals were time to market, cost, and performance. We were not willing to let performance issues significantly impact an aggressive schedule, nor significantly increase cost over a dumb frame buffer system. We wanted to maximize the performance of our cheapest graphics systems.

To minimize design time, we kept things simple. Every piece of logic had to result in concrete performance improvements. We kept functionality as general as possible to allow extensive sharing of common logic among the different hardware modes, and to allow software to use these modes across a variety of painting algorithms.

We wanted to keep board manufacturing costs at or below that of a dumb frame buffer system, so the gate array cost had to be offset by the elimination of random glue logic. The cheapest gate array available had too few pins for a 64-bit data path to video memory. We settled for the next cheapest, with 184 I/O pins and 54,000 gates. We used 22,000 gates, which the manufacturer's router could barely handle. We had enough pins and gates to implement the capabilities we really wanted, and no more. These constraints provided us with a

technical excuse for avoiding additional capabilities that, while desirable, would have significantly lengthened the design time.

To get high performance, we carefully divided responsibility between the sfb chip and the CPU, so that each chip gets to do what it is best at. The gate array extracts the maximum possible bandwidth from the video RAMs; the CPU implements painting algorithms.

Although many graphics accelerators include extensive control logic, we'd rather exploit the capabilities of CPUs than compete with them. The Alpha CPU in a Flamingo workstation ticks at 6.7 nsec – nearly six times faster than our 40 nsec gate array clock – and faster CPUs are on the horizon. And by improving software painting algorithms, we can also increase performance without redesigning the graphics hardware.

We use three strategies to maximize bandwidth and avoid reads and read/modify/write operations over the TURBOchannel. The sfb chip is closely coupled to video memory with a wide data path, and implements semantics for planemasking and the Boolean combination of source and destination pixels. The sfb allows the processor to use 32-bit writes to word-aligned addresses, and so avoid partial word writes that might not be supported by the CPU's instruction set. Finally, all sfb operations complete within a bus timeout, so the processor never needs to check for overflow of the chip's input buffer.

3. System Architecture and Interfaces

The primary external control functions of the smart frame buffer chip are to interface to the TURBOchannel I/O bus, to interface to the random-access and serial ports of the video RAM, to generate timing signals for the monitor, and to convert pixels to analog RGB composite video via a Brooktree RAMDAC. Figure 3-1 shows a block diagram of a complete graphics system built around the sfb.

The processor accesses the smart frame buffer via the TURBOchannel, a 32-bit shared data/address bus clocked at 40 nsec (25 MHz). Writes take at least 120 nsec per 32-bit word, yielding a maximum transfer rate of 33 megabytes/second. Reads take at least 160 nsec, yielding a maximum rate of 25 megabytes/second. The sfb chip is almost a write-only device, and can accept 32 bits of data in the minimum 120 nsec bus write cycle. The processor reads data from the chip only to save sfb state when writing console messages, and to copy pixels from the screen into main memory.

To increase bandwidth, the sfb uses a 64-bit interface to video RAM. As long as accesses stay within a 4096-pixel page, the chip can read or write 64 bits of data in 80 nsec. Access to a new page requires an extra 160 nsec, for a total of 240 nsec. Read/modify/write operations like xor require an additional 120 nsec, for a total of 200 nsec for accesses to the same page, and 360 nsec for accesses to a new page.

Video RAMs have a separate output port, fed by one of two large internal shift registers, for sending pixel data to the screen. Each half of a 4096-pixel page can be loaded into one of the shift registers in a few hundred nanoseconds by using a special memory transaction. When there is not enough data left in the shift registers to display the next scanline, the sfb loads one of the shift registers with the next 2048 pixels of data during horizontal blanking. The sfb sends data from the VRAM output port to the Brooktree RAMDAC, which converts the data to an RGB video signal.

4. Smart Frame Buffer Architecture

The smart frame buffer chip sits between the processor and video memory. The sfb chip operates in a 16 megabyte address space, as shown in figure 4-1. Most of the address space is devoted to the frame buffer memory. The maximum frame buffer size is 8 megabytes, for use in a true color system with up to 1600x1280 32-bit pixels. Since the usual frame buffer size is 2 megabytes of 8-bit pixels, and since early workstations limited TURBOchannel address space, we alias portions of frame buffer memory to fit into smaller 4 and 8 megabyte address spaces.

4.1 Dumb frame buffer mode

The sfb operates in several modes. In the simplest mode, the sfb acts like a dumb frame buffer. The processor can read or write a 32-bit word to any address in frame buffer memory. If the processor architecture supports byte or other partial word addressing, as do the MIPS R3000 and R4000, the processor can read or write any group of bytes within a 32-bit word.

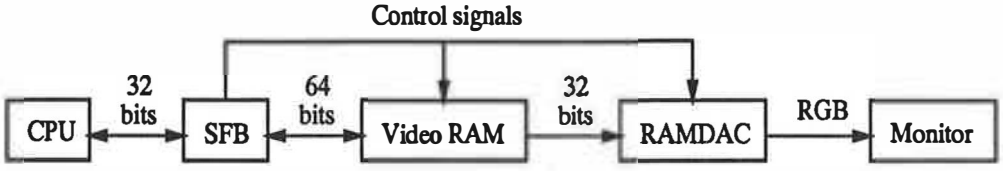


Figure 3-1: Block diagram of primary sfb chip interfaces

4.2 Planemasking and Boolean functions

Even dumb frame buffer mode (and all the accelerated modes described below as well) provides some specialized graphics functionality. The sfb implements a planemask and the 16 possible Boolean functions ("rasterops") for combining source and destination pixels. These operations would otherwise require read/modify/write cycles in all but the simplest cases.

Conceptually, a planemask contains the same number of bits (or "planes") as a single pixel. A 1 in the planemask allows the corresponding bit in the destination pixel to be overwritten, a 0 in the planemask leaves the corresponding destination bit unchanged. Whenever the processor loads the sfb's planemask register, or the sfb accesses a new page, the sfb issues a special cycle to video memory to load the planemask into the VRAMs. The VRAMs then use the loaded planemask as a write-enable bit mask on subsequent writes.

The X protocol allows a source pixel and a destination pixel to be combined using any of the 16 possible two-operand Boolean functions. The same graphics function applies to all bits in the pixels. The sfb chip implements all 16 Boolean functions in hardware. The sfb directly overwrites the destination pixels when using one of the four Boolean functions that do not depend upon the destination. For the other twelve functions, the sfb reads the destination pixels, combines them appropriately with the source pixels, then writes the result back to video memory. These destination-dependent operations require an additional 120 nsec over the basic write cycle time, but this is much faster than forcing the processor to read destination data over the bus, combine it with source data using logical operations, then write the result back over the bus.

4.3 Accelerated mode philosophy

A typical graphics accelerator accepts commands like "paint a rectangle," "paint a triangle," "paint text," and "copy a rectangle." The accelerator executes a sequence of microcode for each command. Each microcode routine computes the location of the object in video memory given its x and y coordinates, computes the shape of the object, clips the object to the window, figures out what data to fill the object with, and then issues a sequence of span filling operations to the most primitive layer of painting logic. (A span is a contiguous sequence of pixels on one scan line.) In many cases, the graphics accelerator chip is more complex and expensive than the processor chip to which it is attached!

The sfb can't even fill a span by itself. It is "smart" only when compared to a dumb frame buffer.

For accelerated painting operations, the processor writes to a few sfb registers, like the foreground and background pixels and the mode register, then writes 32-bit data words into the frame buffer. Each write address is aligned to an 8-byte boundary, and tells the sfb where in the frame buffer to start painting. The write data tells the sfb what to paint. Each bit specifies what happens to one pixel, so a single data word may affect as many as 32 pixels. Different modes cause different interpretations of the 32-bit data word.

Though the smart frame buffer is not much more complex than a dumb frame buffer, it offers several performance advantages. Since the processor uses only one bit per pixel, a system with 8-bit pixels reduces the number of bus transactions by 8 to 16 times. (Some operations in a dumb frame buffer require two transactions per word, thus the factor of 16.) This compaction in turn effectively increases the capacity of the processor's write buffer.

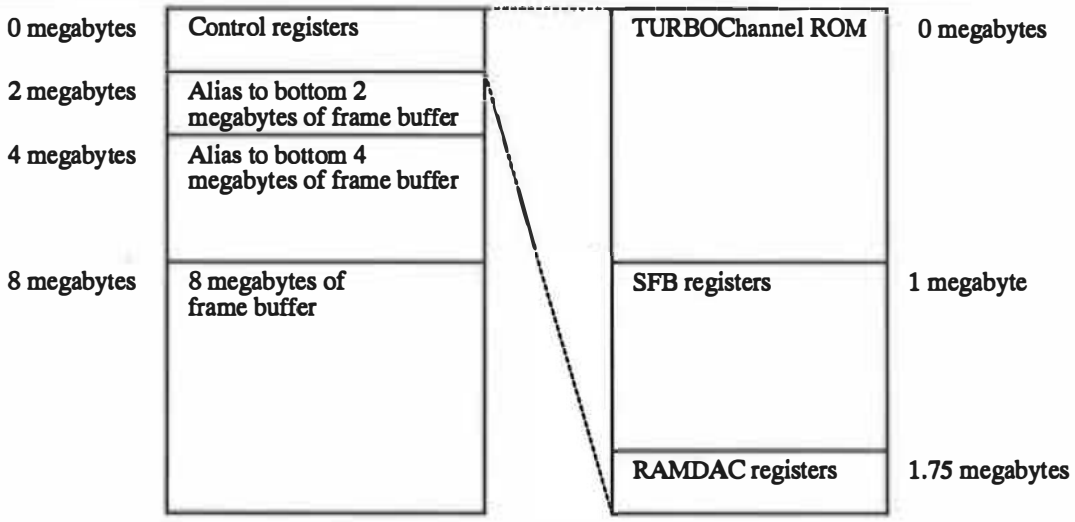


Figure 4-1: Address space of sfb chip

The sfb can write eight 8-bit pixels every 80 nsec. To process a complete 32-bit data word, the sfb normally uses four cycles, or 320 nsec. For most operations, there are no idle cycles spent between 32-bit data words. This results in a measured write bandwidth of 90 megabytes/second – nearly three times the 32 megabytes/second we've measured over the TURBOchannel. We also get small-scale parallelism: while the sfb is processing one data word, the processor can be computing the next word.

4.4 Transparent stipple mode

Transparent stipple mode expands 32 data bits to 32 pixels, with the following semantics:

- 0 means do nothing
- 1 means use the foreground pixel as the source pixel

Figure 4-2 shows a portion of a transparent stipple operation. Transparent stipple mode is used to fill areas with a single color, to fill areas in X11's transparent stipple mode, to paint certain kinds of text, and to fill areas with certain tiles. The sfb has a 32-bit foreground pixel register, which must be loaded before using transparent stipple mode. Software replicates the foreground pixel to 32 bits on 8-bit and 16-bit pixel systems.

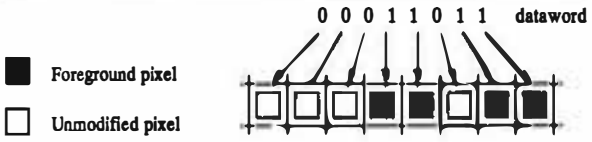


Figure 4-2: Transparent stipple behavior

The left edge of a span may not be aligned to 8 bytes, and the width is rarely a multiple of 32 bytes. The processor uses the no-op property of 0 to deal with these ragged edges. It zeroes as many as 7 low-order bits of the data word it uses at the left edge of a span, and as many as 31 high-order bits at the right edge. To fill a span of less than 32 pixels, it zeroes the appropriate bits at both ends of the data word. The sfb hardware uses a priority encoder to skip over low-order zeroes, and stops painting when only zeroes remain in the high-order bits of a word.

Some graphics chips implement transparent stipple operations using read/modify/write cycles. The sfb avoids reads by using control logic on individual VRAM chips to disable writes to pixels with a 0 data bit. The theoretical peak fill rate is 8 bytes every 80 nsec, or 100 megabytes/second.

4.5 Opaque stipple mode

Opaque stipple mode expands 32 data bits to 32 pixels, with the following semantics:

- 0 means use the background pixel as the source pixel
- 1 means use the foreground pixel as the source pixel

Figure 4-3 shows a portion of an opaque stipple operation. Opaque stipple mode is used to fill areas with X11's opaque stipple mode, to paint certain kinds of text, and to implement `CopyPlane` requests.

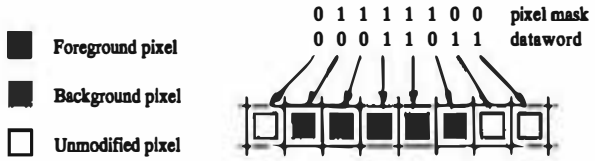


Figure 4-3: Opaque stipple behavior

Like the foreground register, the background pixel register is 32 bits wide. Both foreground and background must be loaded before using opaque stipple mode.

To fill narrow spans, or the left and right edges of longer spans, 0 bits in the data can't be used as no-ops. The sfb provides a 32-bit pixel mask register; a 1 in the mask allows the corresponding pixel to be written, and a 0 prevents the pixel from being written. To write less than 32 pixels in opaque stipple mode, the processor first writes to the pixel mask register, then writes a data word to the frame buffer. The pixel mask register resets to all 1's after each use: most algorithms paint a scanline at a time, so this saves us from writing a mask of all 1's to paint the middle of large spans.

Transparent and opaque stipple modes share large amounts of gate array logic. They differ only in their use of the pixel mask register. Opaque stipple mode uses the pattern that is already in the pixel mask register; transparent stipple mode loads the data word into the pixel mask register. Both modes expand 1 bits in the data word to the foreground pixel, and 0 bits to the background pixel. But transparent stipple mode doesn't paint the background pixels, because the pixel mask register contains zeroes in those positions. The priority encoder and zero-detection logic use whatever pattern ends up in the pixel mask register, which allows copy mode (described below) to use this logic as well. The theoretical peak fill rate for opaque stipples is 100 megabytes/second.

4.6 Copy mode

When copying pixels from one area to another, the sfb cannot synthesize the source data from background and foreground pixels, but must read source data from memory. The sfb includes a 32-byte on-chip copy buffer for temporarily holding source data.

The processor transfers pixels in groups of 32 bytes by writing a pair of 32-bit data words. The processor first writes a data word at the address of the source pixels. A 1 in the data word indicates that the corresponding pixel should be read into the copy buffer, a 0 indicates that the pixel isn't needed. The processor then writes a second data word, this time to the address of the destination pixels. A 1 in the data word indicates that the corresponding pixel

in the copy buffer should be written, a 0 indicates that the destination pixel should be left unchanged.

The sfb requires source and destination addresses to be aligned to 8 bytes, but an application can specify copies that start at arbitrary byte addresses. To support unaligned copies, the sfb uses an 8-byte residue register and a shifter to assemble data from two consecutive aligned 8-byte source words into an aligned 8-byte destination word. This logic works for both forward and backward copies. The residue register maintains data between each 32-byte group of pixels, so that once an unaligned copy is started, each pair of data words copies a full 32 bytes of data.

Assuming that the source and destination addresses are on different VRAM pages, the copy logic has a theoretical maximum bandwidth of 33 megabytes/second.

The on-chip copy buffer is available to the processor as eight 32-bit registers. To transfer data from main memory to VRAM, the processor writes these registers, then writes a 32-bit data word to the destination address in the frame buffer. Conversely, to transfer data from VRAM to main memory, the processor writes a 32-bit data word to the source address in the frame buffer, then reads the copy buffer registers. The residue register and shift logic are enabled in both cases.

The sfb's copy logic illustrates the advantages of keeping graphics hardware simple. We concentrated on making the underlying copy functionality complete – supporting backward copies as efficiently as forward copies, and using the copy logic for transfers between main memory and VRAM – rather than putting higher-level control into hardware by supporting rectangle copies.

Implementing rectangle copies in hardware is a nightmare: overlapping rectangles may require copying from top to bottom or vice-versa, and from left to right or vice-versa, and source and destination addresses may not be aligned to VRAM words. In a vain attempt at simplification, some graphics chips read source data multiple times during unaligned copies. If the sfb took this approach, it would read 32 bytes, then write 24 bytes, slowing unaligned copy rates by 17%. Some chips support unaligned copies from left-to-right, but leave the backward direction to software! And even when a complex accelerator provides full rectangle copy support, it may have bugs – we know of one accelerator that could not copy rectangles of width 1. Had this bug not been circumventable in software, another pass of the chip would have been required.

4.7 Line modes

Transparent and opaque stipple modes paint up to 32 pixels in a horizontal span. To paint a longer span, the processor explicitly provides the starting address of each 32-pixel chunk. Transparent stipple and opaque stipple line modes differ from the corresponding span modes in that the sfb traces out a line that may go in any direction, it paints 16 pixels at a time, and it keeps track of the current address across 16-pixel chunks. Figure 4-4 shows a portion of both transparent and opaque stipple line operations.

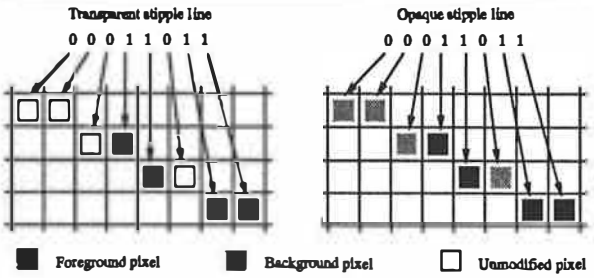


Figure 4-4: Line stipple behavior

The sfb computes the path of a line through frame buffer memory using Bresenham's algorithm [2]. The C equivalent of the hardware Bresenham step looks like:

```
*address = foreground;
if (e < 0) {
  address += a1;
  e += e1;
} else {
  address += a2;
  e -= e2;
}
```

To paint a line, the processor provides initial values for `e` (a signed 17 bit number), `e1` and `e2` (unsigned 16 bit numbers), `a1` and `a2` (signed 16-bit numbers), and the length of the line modulo 16. The processor then writes a data word to the starting address of the line, aligned to a 4-byte word. The data contains up to 16 bits of transparent or opaque stipple line data, and the low two bits masked off the true starting address in order to align it. To paint longer lines, the processor writes 16-bit data words to a continuation register; each write causes the sfb to paint 16 more pixels.

At the end of a line, the sfb leaves the address register one position past the last pixel painted. When painting lines that are connected end-to-end, this is the starting point of the next line. The processor thus avoids a multiply to compute the new starting address of each connected line.

The processor uses transparent stipple line mode for painting solid lines and dashed lines (alternating foreground with blank space), and opaque stipple line mode for double-dashed lines (alternating foreground and background). Since the processor explicitly provides stipple data for each line, dash patterns may be arbitrarily complex.

We estimate that the theoretical limit for 10-pixel connected lines is 650,000 to 700,000 lines/second.

5 Smart Frame Buffer Configurability

The sfb chip can be used to implement a wide range of graphics systems. It offers multiple pixel depths, a cornucopia of screen resolutions and refresh rates, memory configurations from two to eight megabytes, and can be attached to one or two screens.

5.1 Pixel depths

The smart frame buffer supports pixel depths of 8, 16, and 32 bits. Physical pixel depth is fixed for a given graphics board, as memory must be wired slightly differently in each case. Some Brooktree RAMDACs support the appearance of different depths by allowing control bits in each pixel to specify interpretation of the rest of the bits.

The 8 bits per pixel graphics system uses the Brooktree 459 RAMDAC, which has a 256-entry colormap. Each entry in the colormap contains 8 bits each of red, green, and blue intensity data.

The 16 bits per pixel graphics system is intended for use with the Brooktree 463 RAMDAC, which could be configured on a per-pixel basis to use 4 bits each of red, green, and blue intensity data directly from the pixel, or to use 8 bits of the pixel as an index into one of two 256-entry colormaps. This system would support two bits of overlay planes that are displayed "on top" of normal pixel data.

The 32 bits per pixel graphic system would also use the Brooktree 463. This system could display 8 bits each of red, blue, and green directly from the pixel, or use 8 bits of the pixel as an index into one of two 256-entry colormaps. This system would support 4 bits of overlay planes.

5.2 Monitor resolutions and refresh rates

Digital sells monitors offering resolutions from 640x480 to 1280x1024, using refresh rates from 56 Hz to 76 Hz. We wanted to support all these monitors, and any likely new candidates, so we made the sfb monitor timing generation logic fully programmable.

The sfb uses an external pixel dot clock to generate timing signals for the RAMDAC and video RAMs; this clock's frequency is specific to the monitor's resolution and refresh rate. Programmable clocks were noticably inferior to fixed frequency crystals in image clarity; we suspect this was due to minor instabilities in the programmable clock. We turned the disadvantage of using a different crystal for each type of monitor into a user-friendly feature. We use the dot clock frequency, rather than board jumpers or switches, to automatically determine screen resolution and refresh rate. We support all Digital monitors and most of our competitor's as well, ranging from the VGA format of 640x480 at 60 Hz up to 1600x1280 at 76 Hz.

5.3 Memory configurations

Using 256k by 4-bit parts, the minimum memory configuration requires 16 VRAM chips for a total of 2 megabytes. The standard 8-bit 1280x1024 screen uses 1.25 megabytes of video memory. The remaining .75 megabyte is available for off-screen pixmaps, and the sfb makes it easy to use this memory efficiently. The sfb requires only that screen and pixmap rows be padded to a multiple of 64 bits, so software can use a simple one-dimensional memory allocator for off-screen pixmaps.

The sfb-based HX graphics board has space for an additional 2 megabytes of DRAM, although this configuration is not supported as a product. The X server uses this memory for pixmaps; a four megabyte board has ample memory for full-screen double-buffering applications.

A 16-bit pixel system requires 4 megabytes, or 8 megabytes for full-screen double-buffering. A 32-bit pixel system requires 8 megabytes of memory, which is the maximum allowed, and so full-screen double-buffering isn't possible.

5.4 Multiple monitors

The sfb can drive two monitors simultaneously from a pair of 2 megabyte banks of VRAM. Both monitors share the same dot clock, so they must have the same resolution and refresh rate. This makes it possible to support two independent screens using a single sfb chip, saving board space and manufacturing cost. More importantly, it saves a TURBOchannel slot.

6. Software Algorithms

The sfb-specific X server code borrows heavily from the dumb frame buffer code described in reference [4]. We use the dumb frame buffer code to paint to pixmaps that reside in main memory, so we don't have to limit pixmaps to off-screen video memory. We also used this code as a template for sfb-specific code; in many routines the only significant changes were in the low-level span filling loops. By reusing cfb code so heavily, we were able to get a working X11 server connected to the sfb software simulator in just two months.

6.1 Solid area filling

The simplest operation in an accelerated mode is solid area filling; the example in figure 6-1 shows the basic techniques of mask generation used throughout the sfb code. This code assumes that the planemask and foreground color have already been loaded, that the mode has been set to transparent stipple, and that the span has been clipped to the window boundaries.

If p = 1005_{16} and width = 9, the code computes the masks shown in Figure 6-2. (As the sfb paints from left to right, it uses bits in a data word from low to high.)

To paint a rectangle, the processor computes masks and a starting address outside the main painting loop, then branches into a loop for narrow rectangles that can be painted with one data word, or a loop for wider rectangles that require two or more data words.

6.2 Transparent stipples, opaque stipples, and tiles

The X server uses the solid area code as a template for the routines that paint certain stipples and tiles. Stipples are bitmaps that are expanded using transparent or opaque stipple semantics, while tiles are pixmaps that are copied. The bitmap or pixmap pattern is repeated both horizontally and vertically in order to fill areas larger than the pattern.

The most common stipple data is provided in a bitmap with a width that is a power of 2, like 8, 16,

```
/* Compute starting address of span within frame buffer          */
p = pdstBase + y*drawableWidth + x*SFBPIXELBYTES;

/* Compute how many bytes past 8-byte alignment                  */
align = (int)p & SFBALIGNMASK;

/* Align starting address to 8-byte alignment                    */
p -= align;

/* Convert align from number of bytes to number of pixels        */
align /= SFBPIXELBYTES;

/* Add the number of alignment pixels to the total width         */
width += align;

/* Compute a left mask with low 0's where alignment was needed   */
leftMask = SFBSTIPPLEALL1 << align;

/* Compute a right mask with high 0's past the (extended) width */
rightMask = SFBSTIPPLEALL1 >> (-width & SFBSTIPPLEBITMASK);

if (width <= SFBSTIPPLEBITS) {
    /* Mask fits into a single word                              */
    SFBADDRESS(sfb, p); /* Minimize TLB misses                   */
    SFBSTART(sfb, leftMask & rightMask);
} else {
    /* Mask requires 2 or more words                             */
    SFBWRITE(p, leftMask);
    width -= 2*SFBSTIPPLEBITS;
    while (width > 0) {
        p += SFBSTIPPLEBYTESDONE;
        SFBWRITE(p, SFBSTIPPLEALL1);
        width -= SFBSTIPPLEBITS;
    }
    SFBWRITE(p+SFBSTIPPLEBYTESDONE, rightMask);
}
```

Figure 6-1: Solid filling prototype code

```
leftMask              11111111 11111111 11111111 11100000
rightMask             00000000 00000000 00111111 11111111
leftMask & rightMask  00000000 00000000 00111111 11100000
```

Figure 6-2: Example masks

or 32. The most common tile data is provided in a pixmap with a width of four pixels, or 32 bits on an 8-bit pixel system. The sfb code replicates any such bitmap or pixmap to a width of 32 bits, and provides special routines for painting these patterns. These special cases of stipple and tile painting are so similar that the same source code is compiled three times, with a few `#ifdef` statements to implement the differences.

In the transparent and opaque stipple code, the processor fetches a single 32-bit word from the appropriate row of the bitmap, rotates this word based upon the position in the window, then writes the rotated data every 32 pixels across the entire span (masking off a few bits at the edges).

The tile code rotates data on pixel boundaries rather than on bit boundaries, then loads the foreground register with the rotated data. The foreground register is 32 bits wide, so it can hold a different 8-bit pixel value in each byte. The server then fills the span as if it were filling a solid area. Though this code can paint tiles that are no wider than four pixels on an 8-bit pixel system, this is often sufficient. For example, the Display PostScript System [3, 1] uses a tile four pixels wide by six pixels high for color half-toning.

Stipples of widths that are not a power of two are uncommon, so the server code for them is fairly inefficient. The server fetches either a full 32-bit word, or whatever is left of the stipple, then paints this data word. To satisfy alignment constraints, the server usually has to paint the data word in two separate operations; in opaque stipple mode this requires two separate writes to the pixel mask register.

The server uses code similar to that described below for copies in order to fill areas with tiles that are larger than 32 bits in width.

6.3 `CopyPlane`

The `CopyPlane` operation looks like a non-repeated opaque stipple of arbitrary size. These requests are common enough that the server has special code for large bitmap patterns. Since `CopyPlane` doesn't involve the complications of repeating the bitmap pattern, its inner loop arranges data in order to extract maximum bandwidth from the sfb. This loop maintains the unused bits from the previous iteration, fetches one new 32-bit word, shifts and merges these two words, then writes the resulting data word directly to an 8-byte aligned address. In the middle of a span, each 32 pixels require a single write to the sfb, rather than the four writes used by the general opaque stippling code.

6.4 Copies

Copy code is an obvious extension of the `CopyPlane` code, in which the source bitmap becomes a pixmap. Copies involve two independent frame buffer addresses – source and destination – which may not be aligned. The processor must write the shift amount to the sfb, and may need to prime the shift/residue logic at the beginning of a span, and drain the logic at the end of a span.

When necessary, the processor primes the shift/residue logic by reading an extra 8-byte word before the source data, and drains the logic by reading an extra 8-byte word after the source data. Reading unused data is more efficient than any scheme to explicitly prime or drain the logic. We leave the first 8 bytes and the last 8 bytes of video memory unallocated in order to avoid generating addresses outside of the frame buffer.

6.5 Text

X11 has two types of text painting requests. `PolyText` paints a string of characters using transparent stipple semantics, and so spatters foreground pixels onto the destination. `ImageText` paints a string of characters using opaque stipple semantics, and so fills in the area around characters with the background pixel.

In a fixed-metric font, each glyph (bitmap picture of a character) is the same height and width. In a variable-pitch font, glyphs can be different heights and widths. The server uses different strategies to paint variable-pitch and fixed-metric fonts.

The `PolyText` code for variable-pitch fonts uses transparent stipple mode in the obvious fashion. It looks up the bitmap glyph for each character in the string, and paints one glyph at a time from the top row to the bottom. The corresponding `ImageText` code doesn't use opaque stipple mode, because painting background and foreground simultaneously in these fonts is hard: each glyph must be extended up and down to the overall font height, the space between glyphs must be filled in, and in some fonts information from two adjacent glyphs can overlap (as with an overstrike character). The server avoids these problems by clearing a rectangle of the appropriate size with the background pixel, then calling the `PolyText` code.

The `PolyText` and `ImageText` code for fixed-metric fonts share the same source file, with a few `#ifdefs` to handle masking correctly. Since all glyphs are the same height and width, it is easy to merge information from the same row of several adjacent glyphs.

The processor paints glyphs in groups that are guaranteed to fit into a 32-bit data word, regardless of alignment constraints. For example, if each glyph is 6 bits wide, the processor can fit data from four glyphs into a 32-bit data word, and still have room to shift the data left as much as 7 bits in order to satisfy the 8-byte alignment constraint. Similarly, the processor can fit data from three 8-bit wide glyphs into a data word, and still have room to shift the data to satisfy alignment constraints.

6.6 Lines

Though largely irrelevant for most 2D applications, the most commonly quoted graphics performance benchmark is 10-pixel lines. Not coincidentally, line painting is the only area where we descended into assembly code, and literally counted every instruction. We maximized performance by avoiding data shuffling and masking, by using fast clipping code, and by using position within code rather than data registers to record important decisions.

We chose the formats of the line initialization registers in order to minimize the number of writes to the sfb. We then arranged fields within the registers in order to avoid masking operations in the CPU as we shifted and merged data into the proper positions.

To determine if a line is completely visible within a window, we borrowed Keith Packard's code from the MIT X11R5 sample server. This code simultaneously compares 16-bit x and y coordinates in a single 32-bit subtract. Testing unconnected lines for visibility requires 11 instructions. The connected line code remembers visibility information of the ending point, which becomes the starting point of the next line. If this point is known to be visible (the usual case), testing the new end point uses only 8 instructions.

Finally, rather than painting all lines with the same loop, our code branches into one of four cases depending on whether the line is more horizontal than vertical, and whether the line goes forward or backward. This reduces line setup by a few more instructions.

7. Performance Measurements

This section presents a few results from the `x11perf` X server benchmarking program. More complete performance results, with more up-to-date Flamingo numbers, can be found in reference [5].

In order to compare "apples to apples" where possible, we present several different sets of sfb performance numbers with performance results from Sun and Hewlett-Packard machines.

The Sun results are from a SPARCstation 2 with a GX graphics accelerator [6]. This configuration has CPU performance comparable to a DECstation 5000/200. Using the 1992 SPECint benchmarks, the SPARCstation rates about 22 integer SPECmarks, the DECstation about 20. Both use UNIX sockets for communication between the application and the X server.

The DECstation 5000/240 and the Alpha-based Flamingo workstation bracket the HP 730's CPU performance. The DECstation 5000/240 rates about 27 integer SPECmarks, the HP 730 about 48, and the Flamingo about 74. The Alpha performance numbers below are preliminary, and may improve with better compiler technology and with server performance tuning. All three systems use shared memory for communication between an application and the X server.

Benchmark	Sun SS2 w/GX	DEC 5000/200 w/sfb	DEC 5000/240 w/sfb	HP 730 w/CRX	DEC Flamingo w/sfb
Solid 10x10 (krecs/sec)	150	156	225	278	397
Transparent 10x10 (krecs/sec)	150	96	166	136	422
Opaque 10x10 (krecs/sec)	150	96	141	136	296
Tile 10x10 (krecs/sec)	150	93	137	132	295
Solid 500x500 (megabytes/sec)	98	90	91	110	94
Transparent (megabytes/sec) 500x500	98	90	91	69	94
Opaque 500x500 (megabytes/sec)	98	85	86	69	91
Tile 500x500 (megabytes/sec)	97	90	90	69	93
`CopyPlane` 500x500 (megabytes/sec)	50	67	81	65	88

Table 7-1: Rectangle fill performance

Benchmark (megabytes/second)	Sun SS2 w/GX	DEC 5000/200 w/sfb	DEC 5000/240 w/sfb	HP 730 w/CRX	DEC Flamingo w/sfb
Screen to screen	18	32	32	40	32
`PutImage`	1.6	2.9	6.0	6.5	9.0

Table 7-2: Rectangle copy performance

Small rectangles rates may be limited by the CPU or by graphics hardware, while large rectangles show the raw graphics memory bandwidth available. The Sun has identical rates for solid, stippled, and tiled rectangles, as the `x11perf` patterns fit into the GX's stipple memory. But note that an Alpha, which has CPU cycles to burn, easily outdistances both the GX and the CRX painting small rectangles. In fact, the Flamingo fills solid 10x10 rectangles at 89% of the sfb's theoretical maximum of 444,000 rectangles/second.

Table 7-2 shows raw copy bandwidth. Screen to screen copies occur when a window manager moves windows around, or when an application scrolls data within a window. Standard X11 `PutImage` copies occur when an application forms an image in its own memory, then copies the image to the screen.

Text painting performance provides some interesting contrasts between simple and complex hardware. The Sun GX provides facilities for painting text that are similar to the sfb's stipple modes, but involve more overhead, and so Sun's performance is quite low. (But note that an entire page of text contains less than 10,000 characters, so any of these devices would fill a window within a few screen refresh times.)

The X11 `Polyline` request paints lines that are connected end-to-end; *n* lines require *n+1* points. The `PolySegment` request paints lines that are not connected; *n* lines require *2n* points. Connected lines require less data to be copied, may use more efficient clipping and fewer multiplications in the server, and may require less setup in the hardware.

The sfb's transparent and opaque line modes make it easy for software to paint dashed lines quickly; the GX treats dashed lines as a special case -- for which it has no acceleration facilities. We suspect that we are competitive with HP on dashed lines only because of superior software, not because of any hardware limitations.

8. Conclusions

Complex accelerators have traditionally been used to achieve high graphics performance. When designed competently and quickly, they can still provide leading-edge performance. But graphics accelerators are often designed with insufficient understanding of their intended uses, in competition rather than in cooperation with the associated CPU, without adequate consultation with software engineers, and with three-year design cycles. Such accelerators may have worse performance on many operations than a simple dumb frame buffer, but people will use them anyway in the mistaken belief that high performance requires complexity.

We believe that RISC architectures have introduced a qualitative change in the relationship between graphics accelerators and general-purpose processors. It is no longer necessary to put extensive control logic into an accelerator. Such functionality can migrate back to the CPU and software; hardware designers can then focus on providing maximum memory bandwidth to the processor with minimum hardware complexity.

The smart frame buffer provides bandwidth in simple, general ways, rather than trying to trade off chip real estate among several specialized functions.

Benchmark (kilochars/second)	Sun SS2 w/GX	DEC 5000/200 w/sfb	DEC 5000/240 w/sfb	HP 730 w/CRX	DEC Flamingo w/sfb
`PolyText` Times Roman, 10 pt	111	174	276	391	424
`ImageText` Times Roman, 10 pt	89	161	220	320	330
`PolyText` Times Roman, 24 pt	70	81	146	210	221
`ImageText` Times Roman, 24 pt	50	64	91	130	123

Table 7-3: Text performance

Benchmark (kilolines/second)	Sun SS2 w/GX	DEC 5000/200 w/sfb	DEC 5000/240 w/sfb	HP 730 w/CRX	DEC Flamingo w/sfb
10-pixel lines	226	325	587	911	638
10-pixel segments	157	261	461	648	507
10-pixel dashed segments	47	186	360	318	520
500-pixel segments	9	17	17	25	17

Table 7-4: Line performance

This single-mindedness led to a design whose core logic is shared among all modes, which allowed us time to refine implementation details: fine-tuning register formats, eliminating pipeline bubbles and idle cycles, making functionality more complete. Our attention to detail is paid back continually, as the chip's basic functionality is exploited again and again to paint different types of graphics objects.

Leaving control flow decisions in the CPU has one major disadvantage: no large-scale parallelism can take place. The advantages are numerous. The sfb provides a good deal of small-scale parallelism, and graphics performance increases as processors get faster. The simplicity of the sfb decreases development time, so that new designs can closely track the latest capabilities of VRAM technology. The functionality fits into a cheap gate array, so that graphics acceleration adds nothing to the manufacturing cost. Last, but not least, the smart frame buffer offers performance that is comparable, and in many cases exceeds, more traditional accelerators – without suffering "Achilles heel syndrome" when confronted with a case that the designers didn't have the chip real estate, or foresight, to include.

We see a long life for smart frame buffers. We are already working on a chip that will provide higher performance by exploiting a new generation of video RAMs, and will add video and low-end 3D capabilities. Except for high-end 3D systems, we believe complex graphics accelerators are an evolutionary dead end.

9. Availability

Reference [4] is available as WRL Research Report 91/1. More detailed information on the smart frame buffer can be found in WRL Research Report 93/1, "A Smart Frame Buffer". To order, send electronic mail to wrl-techreports@decwrl.dec.com or decwrl!wrl-techreports. Include your name and address, and the line "Order 91/1" or "Order 93/1". Or send a request via U.S. mail to Technical Report Distribution, DEC Western Research Lab, 250 University Avenue, Palo Alto, CA 94301. `x11perf` is available via anonymous ftp from expo.lcs.mit.edu:/contrib/x11perf.tar.Z.

10. Acknowledgements

Lindsay Gage did most of the translation of the high-level behavioral model into schematic diagrams. Chris Gianos designed the sfb-based HX TURBOchannel board. Jim Peacock filled the HX ROM with diagnostic and console code. Jim Gettys wrote the sfb kernel driver. Keith Packard had useful advice about the sfb ddx code.

John Ousterhout critiqued an early draft of this paper.

11. References

[1] Adobe Systems, Inc. PostScript Language Color Extensions. Adobe Systems, Inc., Mountain View, CA, 1988, 1989.

[2] J. E. Bresenham. Algorithm for Computer Control of a Digital Plotter. IBM Systems Journal 4(1):25-30, 1965.

[3] Christopher A. Kent. XDPS: A Display PostScript System Extension for DECwindows. Digital Technical Journal 2(3):64-73, Summer, 1990.

[4] Joel McCormack. Writing Fast X Servers for Dumb Color Frame Buffers. Software: Practice and Experience 20(S2):83-108, October, 1990.

[5] Joel McCormack. A Smart Frame Buffer. WRL Research Report 93/1, Digital Equipment Corporation, Western Research Laboratory, January, 1993.

[6] Curtis Priem. Sun GX Graphics Workstations, The Standard for Graphics Performance from the Desktop to Powerful Deskside Systems. In Symposium Record. Hot Chips Symposium, June, 1989.

12. Author Information

Joel received a B.A. in 1978, and an M.S. in 1979, from the University of California, San Diego, where he worked for the UCSD Pascal project. He cofounded Volition Systems, which sold Modula-2 compilers. He joined DEC's Western Software Laboratory in 1984 to write a Pascal compiler for the Titan RISC processor, then became lead architect for the X11 toolkit intrinsics. Joel joined DEC's Western Research Laboratory in 1988, where he has mostly worked on graphics software and hardware. His e-mail address is joel@decwrl.dec.com or decwrl!joel. His mail address is DEC Western Research Laboratory, 250 University Avenue, Palo Alto, CA 94301.

Bob McNamara received a B.S.E.E. from the University of Michigan in 1979, and went directly to Digital, where he designs workstations and graphics hardware. Reach him electronically at mcnamara@pa.dec.com or decwrl!mcnamara. His mail address is DEC Smart Frame Buffer Group, MLO3-6/C9, 146 Main Street, Maynard, MA, 01754.

Wafe – An X Toolkit Based Frontend for Application Programs in Various Programming Languages

Gustaf Neumann & Stefan Nusser – Wirtschaftsuniversität Wien

ABSTRACT

Wafe provides a flexible and easy to use interface to the *X Toolkit* (Xt) and the *Athena* widget set (Xaw) using the embeddable command language *Tcl* [1]. It allows access to Xt's functionality from all compiler and interpreter languages, provided that they can communicate over *stdout* and *stdin* via unbuffered I/O. A typical Wafe application consists of a frontend process and an application program, which is executed as a child process of the frontend. Wafe provides a relatively high level interface to the X Toolkit and widget programming, where the user interface can be interactively developed without any need to program in C. Wafe can be used as a rapid prototyping tool and allows easier migration from existing ASCII based programs to X Window applications.

Introduction

When we started to work on the Wafe project in Summer 91 we had the need to provide decent user interfaces for applications in various (mostly interpreted) programming languages. As a matter of fact, at this time most of our applications were running with ASCII based user interfaces under terminal emulators like xterm – which is a practical but suboptimal way of using the graphical user interface of our equipment, which consists mostly of X Window based workstations. We found out that for most (if not even all) of our application programs a small set of X Toolkit commands and the Athena widget library with its programmatic interface was completely sufficient to provide easy to use graphical interfaces.

On the one hand, it seemed impractical to implement widget functionality in all different programming languages used for our applications, on the other hand we did not even consider to port our existing programs to C. Therefore we chose a frontend approach where all widget functionality is incorporated in one separate program. We called our frontend Wafe, standing for *W*idget[*A*thena]*F*ront-*E*nd. Wafe was implemented using the embeddable command language Tcl [1], which was augmented with widget specific facilities. Tcl is an interpretative language using strings as the only data type and provides a collection of built-in utility commands as well as user defineable subroutines.

Given the situation described so far, we decided after a thorough analysis of the existing products to implement our own solution using the following design goals:

- Our frontend approach must be able to collaborate with a broad variety of programming languages, using a handy communication mechanism. This implies that we cannot presume that the backend application will support certain libraries (e.g., sockets or pipes), which are actually not available under some of the programming languages used for the examples presented in the last section.
- To support smoothly the different stages of the developing and prototyping process, we want our frontend application to provide three different modes of operation: There is an *interactive mode*, where Wafe can be used as a single process reading commands from standard input, which are interactively interpeted. The user sees how the widget tree is built and modified step by step. The interactive mode offers the possibility to examine the effects of different commands or to easily compare different approaches to accomplish a certain task.
  Furthermore, our frontend has to support the possibility to execute command files (*file mode*). The file mode offers two main usages: First, this mode can be used to provide simple user interfaces just by writing scripts in the Tcl language, where Tcl's built-in commands or the commands provided by Wafe can be used. Typically such a script will start with the `#!` magic supported by most of the shells. This script can also be used later as a frontend. The user interface (the frontend) can be developed mostly independent from the application program (the backend).
  Finally, Wafe provides the so-called *frontend mode* which uses interprocess communication facilities as described in the sections below to support the separation between the backend application and the frontend process.

- Another requirement for the application development was the extensibility of the chosen widget set. This made us choose the X Toolkit as the basis for our program, granting access to the broad range of commercial or freely available widgets using this toolkit.
- Finally, as mentioned above, we want to use Wafe as a prototyping tool as well, for developing and testing applications which will be implemented finally in another programming language (mostly C). This requires the incorporation of a widely available widget set. We chose the Athena widgets as the basis for our project, since they are part of the MIT standard distribution of the X Window System. Accepting that they do not offer a very exciting appearance, a version supporting the commercial OSF/Motif widget set is under development (at the time of this writing).

The first section starts with a short comparison of Wafe and Tk [2] which was one of the ancestors (motivations) of the Wafe project. The following section presents an overview of the components, followed by a summary of the design principles and basic features of Wafe. After that we will discuss how Wafe can be used as a frontend for application programs in arbitrary programming languages. This section contains a programming example in Perl [3]. The summary of our experiences and an availability note end this paper.

Comparison between Wafe and Tk

The regular USENIX conference visitor who is confronted with the terms "Tcl" and "user interface" will associate immediately John K. Ousterhout's work on Tcl [1] and Tk [2]. Therefore we want to give a short comparison of Wafe and Ousterhout's work before we concentrate on the details of Wafe.

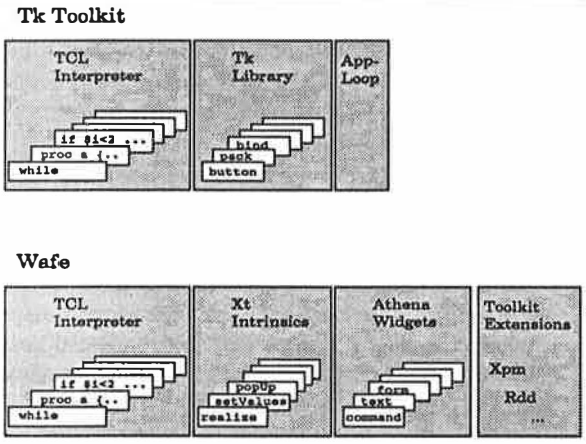


Figure 1: Tk and Wafe components

In some of its components Wafe looks similar to the Tk toolkit [2]. Tk comes with a Tcl shell (*wish*) which allows to read in command sequences in Tcl from a file. Wafe's equivalent is the file mode mentioned above.

The Tk intrinsics and Tk widgets have been implemented by John Ousterhout since 1989; Tk offers three dimensional appearance of its widgets, its implementation compares favorably with the Motif counterparts in terms of size (see [2]).

Wafe is – on the other hand – based on the standard X11R5 Xt Intrinsics [4] and the Athena widget set [5] (see Figure 1). As a consequence it was easy to extend Wafe with other Xt based widgets, widget sets or libraries such as Xpm [6] or for example a drag and drop library (Rdd [11]). A user interface designer can use the standard X11R5 literature (knowledge, support) in order to develop Wafe applications. It is straightforward to replace the Athena widgets by any other Xt based widget set (such as Motif) or to augment Wafe with special purpose widgets. The current Wafe distribution contains support for the Plotter widget set (which supports bar graphs and line graphs [12]) and the *XmGraph* widget (a graph layout widget for OSF/Motif used in Figure 2 [13]). Kaleb Keithley's three dimensional Athena widget library (Xaw3d) [10] can be used simply by relinking Wafe.

In order to write larger applications in practically arbitrary languages, Wafe provides its frontend mode. Current versions of Tcl and Tk do not provide any comparable facility.

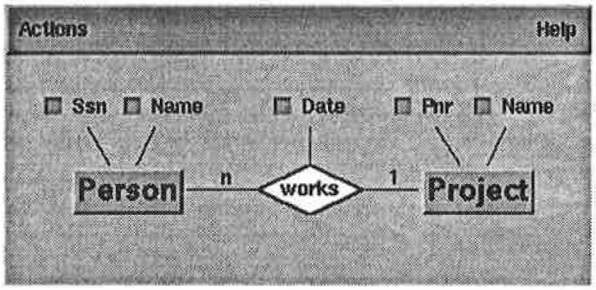


Figure 2: Sample Wafe application using the *XmGraph* widget based on OSF/Motif

The Components of Wafe

This chapter is intended to present the most important implementation issues and to explain some design principles which build the basis of all user level commands.

Wafe's structure can be described globally by the following formula:

Wafe = Tcl +
(Intrinsics + Widgets + Converters + Ext) +
(Memory Management + Communication)

The three main components of this formula are described in the following sections.

Tcl

Wafe uses the embeddable command language Tcl (first part of the formula) as a host language and extends Tcl's basic programming capabilities with additional X Toolkit and widget specific commands. Tcl provides a parsing mechanism as well as a procedural framework for the generic Wafe commands and offers advantages to users already familiar with other Tcl based tools.

In our point of view Tcl offers the following advantages as a host language:

- Tcl has a simple syntax: Every command is simply a list of words.
- Tcl is highly extensible, since it has a simple and well documented interface to C where each argument is only a string.
- Tcl has a clean memory management, where it is possible to specify whether Tcl should copy variables or where special routines can be specified to free memory.
- Tcl uses only one type of argument – the string. Since the string representation is also used to specify information in resource files, the standard Xt converters can be used to convert from string to the variety of Xt or widget specific data types.

Of course, the use of Tcl imposes some limitations too (see also [2]):

- It is not suitable for more complex programs, since it was designed to be a command language.
- The string representation of all data types is a disadvantage, when repetitious calculations have to be made in Tcl.

Besides *Tcl*, two different groups of components can be distinguished in the *Wafe* formula above. For the following part we assume a certain familiarity with the X Window programming tools, which are extensively described in [7] or [8].

X Toolkit Specific Components

The second unit in the above formula represents all functions and commands actually implementing the programmatic interface to the X Window System. These commands provide access to the functionality of the X Toolkit, which comprises all commands necessary to manage a widget's life cycle, the selection mechanism and some information retrieving functions as well as the basic widget classes. This functionality is commonly called the X Toolkit *Intrinsics*. In most cases, each toolkit function is represented by a corresponding Wafe command.

In addition to the X Toolkit a suitable widget set (*Widgets*) is needed. Examples are the Athena widget set, the OSF/Motif widget set, or the widgets of the Open Look Intrinsic Toolkit (Olit). Every widget set typically has a series specific functions, which are called the "programmatic interface". In general, the whole functionality of the programmatic interface is accessible via corresponding Wafe commands.

Converters are an Intrinsics based concept to set and read resources. The X Toolkit provides mechanisms which allow to register additional application specific converters. Wafe registers several converters to ease the handling of certain resources or to implement some additional functionality. This will be discussed in detail later.

Internals

The third unit of the formula above comprises the necessary internals which put Wafe to work: Since the creation of a widget always implies the dynamic allocation of memory for the associated resources, *Memory Management* is a topic of special importance. Wafe has its own memory management: every time a string resource, a callback - or other objects larger than one word – are updated, the old value is freed. If a widget is destroyed the associated resources in Wafe's memory are disposed too.

The other part of the internals deals mainly with the *Communication* mechanism and its different options, which are described in detail later when Wafe's frontend mode is discussed. Note that most of these internals are hidden from the user, although some commands offer the possibility to extend and customize the communication mechanism.

Design Principles

We tried to present a consistent interface, which is based upon certain design principles. We will present them in the following section.

Transparency

Internals should be hidden from the user as much as possible. Interfaces to the corresponding X Toolkit functions are simplified wherever possible. For example, widgets and even windows are therefore referenced by the widget's name, or as another example, function pointers to certain handlers or callbacks are just executable Tcl string expressions, which are evaluated at the time the handler or callback is invoked.

Naming Conventions

Naming conventions are kept as follows: Wafe commands corresponding to X Toolkit functions (e.g., *XtDestroyWidget*) have the same name except that the prefix "Xt", "Xaw" or "X" is stripped and the first letter of the remaining string is translated to lower case (in our example, the resulting command is called `destroyWidget`). The same principle is applied to all commands associated with the Athena widget set. For example *XawFormAllowResize* is called `formAllowResize` in the Wafe framework). It should be noted that the Athena widget set is the primarily supported library. On the contrary, OSF/Motif commands stripped by the rules

above result in Wafe commands starting with the letter *m*. The OSF/Motif command *XmCommandAppendValue* is therefore called `mCommandAppendValue` in the Motif version of Wafe.

It should be noted that all Wafe commands are generated automatically from a high level description (for code generation, see below). During the code generation the naming rules above are applied. If one prefers other naming conventions, or does not like the prefix stripping at all, it is fairly easy to change the names. In addition Tcl allows to register the same command under various names.

Following the X Toolkit Programming Philosophy

In order to build a frontend Wafe offers commands which should be explained in the X Toolkit documentation. Widgets are created and configured, then the widgets are realized, and during the run of the application the execution flow is triggered by actions and callbacks.

In general one Xt or widget specific call of a C procedure corresponds to one Wafe command. In certain cases, Wafe provides convenience procedures, which group several commands together and help to hide internals.

The *widget creation commands* are an exception from this rule: Instead of implementing one command to instantiate a widget from a certain class (namely *XtCreateManagedWidget*), Wafe provides a different command for each widget class to create an instance. The names of these commands are derived from the corresponding classes in an analogous manner. To create an instance of the Athena Toggle widget class, the command "`toggle` *Name Father*" is provided. In order to create an OSF/Motif *XmCascadeButton*, the creation command is called `mCascadeButton`, and so on.

Command Line Arguments

When command line arguments are passed to Wafe, it has to be determined, for which part of the application these parameter are relevant. In general, there are three candidates:

- The X Window Toolkit,
- the frontend (Wafe), or
- the application program.

We have choosen the following approach: Command line arguments starting with a double dash (like "`--f`") are always handled by the frontend. The remaining arguments are passed to the X Toolkit (to interpret arguments like "`-display hostname:0`" or "`-xrm ...`"). The still remaining arguments are handed over to the application program, if Wafe runs in the frontend mode.

Argument Style and Value Passing Conventions for Wafe Commands

Wafe's argument style is similar to other Tcl based tools: Arguments are separated by spaces, can be grouped as described in [1] and are all of data type *string*. Xt function calls returning a single value are implemented using the standard Tcl method of return value passing. In C programs Xt functions returning several values receive a pointer to a free memory area, where the return value will be placed. The Wafe counterparts of these functions take a name of a Tcl associative array as an argument (instead of a pointer) and create entries in the associative array corresponding to the C-structure's components. We did not have the intention to implement all components of a structure, since some components are rather meaningless in the Wafe context (for example: a display pointer). If an Xt command is used which returns a structure, it should therefore be checked with the Wafe documentation which members are supported. When a C procedure returns a list of a certain type and its length, we return the number of elements as a function value and provide a variable name for the list.

This principle can be illustrated by the following example. The toolkit function `XtGetResourceList` has the following syntax:

```
void
XtGetResourceList(
    WidgetClass,
    XtResourceList* /*return*/,
    Cardinal*       /*return*/
    );
```

The corresponding Wafe function is named `getResourceList`, accepts two arguments (`widget` and `varName`) and returns as function value the number of elements in the list named in the second argument. Since Wafe applications do not deal with the structure *WidgetClass*, we use a widget instance to refer to the class. The string containing the widget name is used to refer to a widget instance and is passed to `getResourceList` as first argument. Therefore `widget` references a previously created widget by its name. The second argument `varName` is the name of the Tcl variable to be created. Let's consider the following example, which can be issued interactively by using Wafe in interactive mode:

```
label l topLevel
echo [getResourceList l retVal]
echo Resources: $retVal
```

The first command creates an instance of the Athena label widget class named `l` as child of the `topLevel` widget (which is a top level shell automatically created in every Wafe program). When the second command is executed, the output of the command between square brackets is printed on standard output. Thus, if the command is executed, the number of resources available for the *Label* widget class is printed, which is 42 using the X11R5 Xaw3d libraries. In addition, a list of the *Label* widget's resources is passed to a variable named in the second argument. A Tcl variable will

be created containing the desired information as a Tcl list structure. In the code example above the third command prints the contents of this variable. (Note that the dollar sign is used for variable substitution in Tcl). The output of the third command looks as follows:

```
Resources: destroyCallback
ancestorSensitive x y width
height borderWidth sensitive
screen depth colormap background
(...)
```

The *XtResource* structure actually contains more members than just the resource's name (such as the default value or the data type for example), but Wafe currently supports only the resource names.

Code generation

As noted above, all Tcl commands provided by Wafe are generated automatically from a high level description. The code generation is performed by a Perl program, which takes as argument the specification file and outputs the necessary C code for conversion, argument passing, error messages, storage management, interpretation of percent codes for callbacks (see section about callbacks below) and registrations of commands. In addition the code generator outputs TeX source for the short reference guide. The main advantages of the code generator are that it (a) provides consistency in documentation and interface code, (b) eases changes that effect code changes on various different places, and (c) makes Wafe easily extensible.

The following example of the specification suffices to provide a `mCascadeButton` command in Wafe.

```
~widgetClass
XmCascadeButton
    include <Xm/CascadeB.h>
```

The specification below creates the Wafe command `mCascadeButtonHighlight` with two input arguments. The command can be used to toggle the state of a OSF/Motif cascade button widget.

```
void
XmCascadeButtonHighlight
    in: Widget
    in: Boolean
```

The Wafe source is currently about 13000 lines of C code. About 60% of the code is generated automatically from specifications like the two examples above. For widget sets with highly regular patterns in their man pages (like OSF/Motif), it is even possible to derive a first draft version of the specification directly from the manual pages.

Basic Features of Wafe

This section presents an overview of Wafe's functionality. In order to obtain a complete documentation of all available Wafe commands refer to the Wafe distribution referenced at the end of this paper.

Creating Widgets

The creation of a widget is certainly the most fundamental task to accomplish with Wafe. It can easily be done using the widget creation commands presented in the last section. Note that these commands correspond to the configuration of a specific Wafe binary – if you choose to install the OSF/Motif version, the command to create the Athena text widget, `asciiText`, won't be available, since in the current version it is not possible to mix Athena and OSF/Motif widgets and converters freely.

All widget creation commands take – after the widget's and the parent's name – any number of attribute-value pairs as additional arguments, which are used to set resources at the widget's creation time.

Consider the following example creating an instance of the OSF/Motif *XmPushButton* widget class under the top level shell:

```
mPushButton pressMe topLevel
```

This command will create a managed *XmPushButton* widget named *pressMe* as a child of the application's top level shell widget. The creation of unmanaged widgets is easily accomplished by an optional argument.

When a Wafe application wants to display widgets on multiple X servers it can create several application shells where the display is specified instead of the father widget.

```
applicationShell top2 dec4:0
```

The children widgets under top2 will be mapped to the specified display.

Setting and Retrieving Resource Values

Resource Values are public variables of a widget instance, which are intended to be set by the programmer or to be configured by the user. Wafe provides several ways to set resource values:

- Using a resource description file, which is evaluated at startup time of the application.
- Using the command `mergeResources`.
- With arguments to the widget creation commands at creation time.
- With the command `setValues` after a widget's creation.

Note that the order of these possibilities to set resource values above corresponds to their precedence. All of the commands will be described in the next paragraphs.

The resource file mechanism

The resource file mechanism, extensively documented in [7], can be used by any Wafe application. Note that Wafe provides some additional converter procedures for the types Pixmap, Callback or XmString. Such resources can be set in the current version of Wafe only during widget creation or via `setValues`.

The mergeResources command

An extension to the resource file mechanism is provided by the Wafe command `mergeResources`. Whenever a widget is created, the per display database of resource specifications is searched for entries relevant for the new widget instance.

By using `mergeResources` the resource database can be extended with additional specifications. The specified resources can refer to widget classes as well as to instances. For short Wafe scripts it is often preferable to have the code as well as the resource specifications together in one file.

This possibility is illustrated by the following example, which could be part of a Wafe script as well as part of a front end application.

```
mergeResources \
   *Font       fixed \
   *foreground blue \
   *background red
(...)
label hello topLevel
```

The resource specifications are used as if they were specified in an application defaults file. The label widget created afterwards will use the three values specified, but they apply as well to every other widget created in this application. The `mergeResources` command can be used at arbitrary places in a Wafe application.

Arguments to widget creation commands

All widget creation commands take any number of additional attribute-value pairs as arguments. Since Wafe uses the standard Xt resource file mechanism in order to convert the specified values to their corresponding data types, you can as well use the features provided by the additional type converters, which will be described below.

Consider the following example, which creates an instance of the Athena `Label` widget class using red background and blue foreground colors.

```
label label1 topLevel \
   background red \
   foreground blue
```

As already explained these specifications override any settings in resource files or settings made with `mergeResources` and therefore reduce the configurability of the application via resource file.

The setValues command

The `setValues` command is used to change a resource value after the widget has been created. Note that there are some resources which cannot be set after creation time or after the widget is realized. For detailed information refer to the documentation of the specific widget class.

In order to change the resource value of `background` and `label` of the previously created widget `label1` the following statement can be used

```
setValues label1 \
   background tomato \
   label "Hi Man"
```

For convenience the command `setValues` is registered as well under the name `sV`.

The Wafe command analogous to `sV` to retrieve values from resources is the command `getValue` (or `gV` for short).

```
echo [gV label1 label]
```

The Wafe command above outputs the content of the label resource of the widget `label1`.

Callbacks and Actions

The X Toolkit provides two mechanisms to link widgets to application code: Callbacks and Actions. Since Wafe's interface is slightly different from the original Xt functions it is described in detail in this section.

Callbacks

Callbacks are used to invoke a function whenever certain predefined requirements are satisfied. Callbacks are defined by the widget itself, which declares a callback resource. An application programmer cannot configure a new callback, she/he can just decide whether to use the callbacks provided by the widget class or not. Actions are more flexible to use since they can be bound to an arbitrary event but they require a more complicated handling.

The most common use of callbacks in Wafe applications will be of the form

```
command hello topLevel \
   callback "echo hello world"
```

where the callback procedure is set via resources. This converter will be discussed in the next section. Using the converter an arbitrary Wafe command can be provided.

In addition to this facility special purpose callback functions offered by the X Toolkit can be used as well. These predefined callback functions can be bound to a widget's callback resource by using the Wafe command `callback`. The different predefined functions available are summarized in the table below:

Predefined Callbacks	
Type	**Description**
none	realize shell, grab none
exclusive	realize shell, grab exclusive
nonexclusive	realize shell, grab nonexclusive
popdown	unrealize shell
position	position shell
positionCursor	position shell under pointer

All of these callback functions concern the handling of popup shells, which are used for menus, dialog boxes and the like. Wafe's access to the predefined callback functions is illustrated by the following code segment for the OSF/Motif version of Wafe, which presumes a previously created *Shell* widget called `popup`.

```
mPushButton b topLevel
callback b armCallback none popup
```

The Motif *PushButton* widget's *armCallback* resource triggers the specified function whenever the button is pressed. In the example the specified predefined function with the name *none* realizes the popup shell `popup` without constraining user events to it.

Actions

Wafe's interface to actions is essentially the command `action`, which is used to override, augment or replace the translation table of a widget with translations specified as arguments. Note that a widget's translation table is actually maintained as a resource called `translations`.

Consider the following example: The Athena *MenuButton* widget provides a simple mean to realize and place a popup shell on a button press. To modify the translations of this widget in order to let the menu pop up whenever the pointer enters the button the following Wafe command can be used:

```
menuButton mb topLevel
action mb override \
   "<EnterWindow>: PopupMenu()"
```

The first command creates an instance of the *MenuButton* widget named `mb`. The second command binds the enter window event to the action `PopupMenu`, which is provided by the *MenuButton* widget class. `PopupMenu` is a built-in action of the X Toolkit.

In addition to the built-in actions provided by Xt and the used widget sets, Wafe provides the possibility to bind the execution of an arbitrary Wafe command to an event. Wafe registers a global action `exec` which accepts any Wafe command as argument. When the action is activated, the Wafe command is executed.

One of the big advantages in using actions instead of callbacks is the possibility to access information from the event which triggered the execution. This feature is supported in a restricted fashion by the `exec` action with printf-like percent codes. The event types supported in this way are:

- Button Press, Button Release
- Key Press, Key Release
- Enter Notify, Leave Notify

Since the information passed to an action depends on the type of event that triggered it, only the following combinations of percent codes and event types are valid:

Event Types and Percent Codes of Actions		
Code	**Information**	**Events**
%t	event type	all of the above
%w	widget	all of the above
%b	number of button	BPress, BRelease
%x	x-coordinate	all of the above
%y	y-coordinate	all of the above
%X	x-root-coordinate	all of the above
%Y	y-root-coordinate	all of the above
%a	ascii-character	KPress, KRelease
%k	keycode	KPress, KeyRelease
%s	keysym	KPress, KeyRelease

It is the programmer's responsibility to ensure by a correct binding in the translation table that a percent code substitution occurs only with a valid event type. The `%t` code will expand to `unknown`, if the event is not included in the list above.

Let us consider as an example an Athena *Label* widget. With the following translation, the key-code, character and keysym will be printed any time a key is pressed in the label widget called `xev`.

```
label xev topLevel
action xev override \
   {<KeyPress>: exec(echo %k %a %s)}
```

If the input "`w!`" is typed on the label widget `xev`, Wafe prints the following output to the associated terminal:

```
198 w w
174 Shift_L
192 ! exclam
```

Converter Procedures

Converters are an Xt Intrinsics based concept which is used to implement conversion for the resources of a widget. In Wafe, a converter always converts a string to a certain target data type; the X Toolkit provides easy mechanisms to provide additional converters.

We tried to use converter procedures whenever we decided to extend the standard Xt mechanism. Some of Wafe's additional converter procedures will be described in this section.

The Callback Converter

We have already introduced Wafe's `callback` command in the last section; the callback converter is used to bind the execution of a Wafe command to a widget's callback resource. Since this feature is implemented as a converter, the standard `setValues` command can be used to set the resource, or the resource can be provided in the resource list in a widget creation command.

The following example shows how to provide the callback resource in a widget creation command

```
command quit topLevel \
    callback quit
```

or to set (or to alter) it later using `sV`:

```
command quit topLevel
sV quit callback quit
```

In this example `callback` is the name of the Athena *Command* widget's callback resource and `quit` a simple Wafe command used to terminate an application.

Some widgets pass additional information to certain callback functions. To access this so-called *clientData*, Wafe uses again printf-like percent codes. Note that these percent codes are only interpreted for certain *Callback* resources in certain widget classes. The complete list of percent codes for each widget class can be found in the Wafe short reference manual. Below is a table of the percent codes for the `callback` resource of the Athena *List* widget class as an example:

Athena List Widget Callback	
Percent Code	**Description**
%w	widget's name
%i	index
%s	active element

The X Toolkit passes the widget pointer referring to the invoking widget to every callback function. This widget pointer is evaluated by using `%w`. Since this information is available for each callback function in Wafe, `%w` can be used in any callback function to obtain the widget's name. The following example shows a statement to set a previously created Athena label widget named `confirmLab` to the selected item of a list widget named `chooseLst`. Selecting an item of a *List* widget activates the specified callback procedure.

```
sV chooseLst callback \
   "sV confirmLab label %s"
```

Opposite to the X Toolkit it is possible in Wafe to obtain the value of a callback resource. The following Wafe script creates a *Form* widget with two *Command* widgets as children. The callback of the second command widget (`c2`) is set to the content of the callback resource of `c1`. When the widget tree is realized and the callback of `c1` is activated, the string "i am c1." is printed; if the callback for `c2` is activated, the output is "i am c2.".

```
#!/usr/bin/X11/wafe --f
form f topLevel
command c1 f \
    callback "echo i am %w."
command c2 f \
    callback [gV c1 callback] \
    fromVert c1
realize
```

The XmString Converter

The Wafe OSF/Motif version provides a converter to *XmString*, which is Motif's compound string data type. A compound string is an extended string format, which additionally contains font information and the string's writing direction. The converter procedure allows to provide compound strings in a user friendly way in a widget creation command or in a `sV` or `gV` command.

Please refer to [9] or any other OSF/Motif book for a complete description of compound strings; the following example using the OSF/Motif XmLabel widget should illustrate the point:

```
#!/usr/bin/X11/mofe --f
mLabel l topLevel \
  fontList \
   "*b&h-lucida-medium-r*14*=ft,\
    *b&h-lucida-bold-r*14*=bft" \
  labelString \
   "I'm^bft bold^ft and^rl strange"
realize
```

The syntax of Wafe's compound string interface is straightforward and similar to TeX's text formatting commands. A special character (we are using "^" instead of TeX's "\") is used for layout commands which are either used to change the font or to change the writing direction. The output of the sample script is shown in Figure 3.



Figure 3: An OSF/Motif widget with compound strings

The Pixmap Converter

The X Window pixmap format (Xpm [6]) is a graphical image file format similar to the standard X11 bitmaps, but it supports colored images and shape masks. Wafe provides an extended String-to-Bitmap converter which checks additionally whether the specified file is in Xpm format, when the attempt to read the file in the standard X bitmap format

failed. This converter can be used to set all resources of type *Pixmap*, such as for example the background pixmap of the Athena *Label* widget.

Using Wafe as a Frontend

In our framework a typical Wafe application consists of two parts, the frontend (Wafe) and an application program, which typically run as separate processes. The application program talks to the frontend via stdio. Each output line from the application process starting with a certain prefix character is interpreted as a Wafe (or pure Tcl) command. So an application program can dynamically submit requests to the frontend to build up and modify the graphical user interface; the application can even down load application specific Tcl procedures to the frontend, which can be executed in the frontend without interaction with the application program. At the same time the application program reads from stdin, which is connected to Wafe, and awaits ASCII strings to control its actions.

Starting Applications in Wafe's Frontend Mode

When Wafe is used in the frontend mode, an application program is started as a subprocess of Wafe. After the fork the necessary connections of the I/O channels are established (see Figure 4, left hand side). Note that in interactive mode or in file mode no subprocess is spawned, and Wafe behaves like a shell.

The first question, however, was to figure out, what application program should be launched as subprocess. Although Wafe provides a command line option to specify the name of the application program, it is in many cases not convenient to be forced to specify this argument. Therefore we chose the following naming scheme:

Suppose an application program is named `wafeApp` (see Figure 4). If a link like `ln -s wafe xwafeApp` is established and `xwafeApp` is executed, the program `wafeApp` is spawned as a subprocess of `wafe` and connects its stdio channels with the frontend.

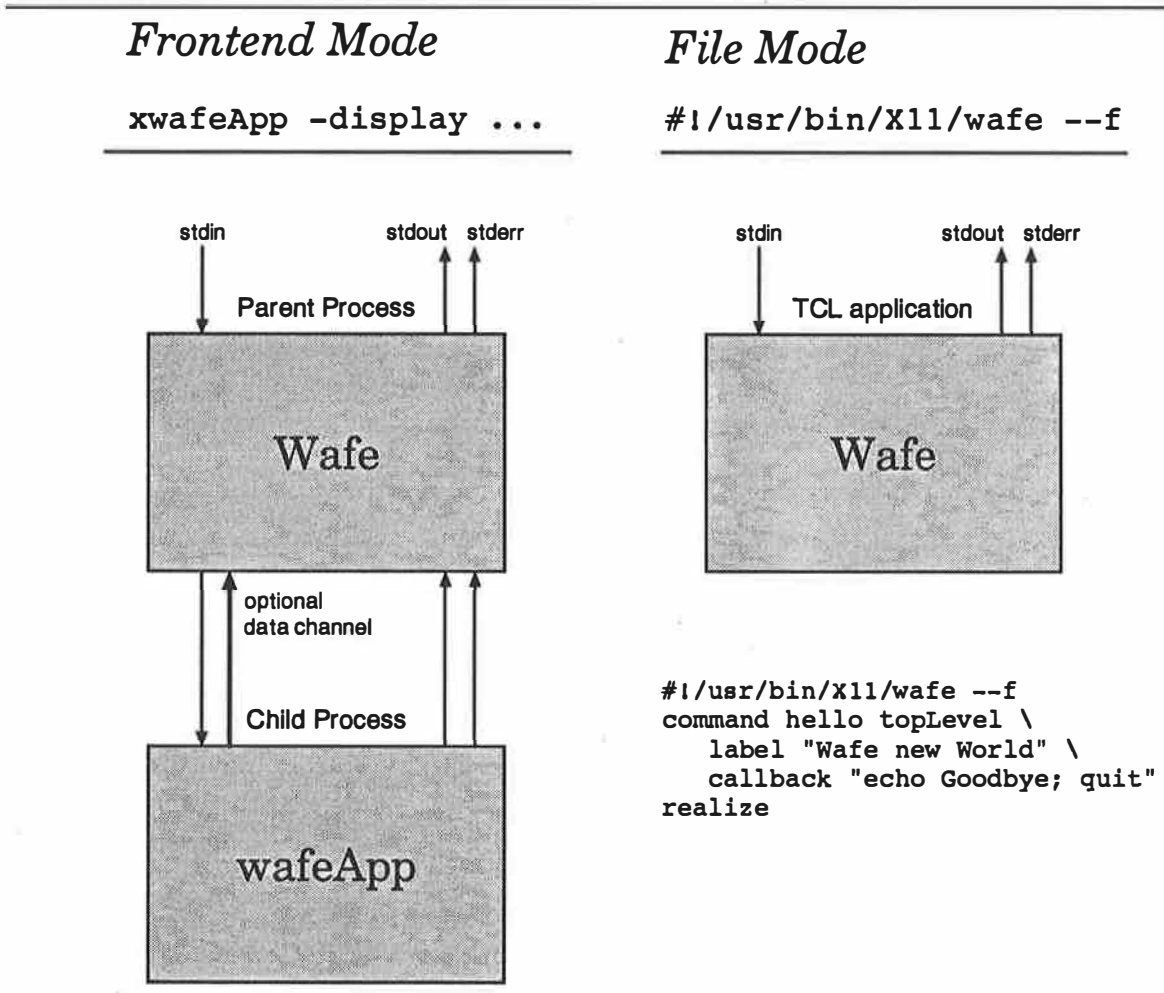


Figure 4: Wafe's Communication Mechanism

Lines written from the application program to *stdout* are read by the Wafe process. If the line received by Wafe starts with a certain character (such as %) Wafe tries to interpret the remainder of the line as a Tcl command. Note that each command issued that way has to fit in a single line (which can be pretty long depending on a preprocessor variable specified at compilation time; the default length is 64KB).

The commands submitted to Wafe can be issued from arbitrary programming languages provided that they are able to write to *stdout* unbuffered (the application program must at least be able to flush the buffer) and to read from stdio. The frontend is programmed by the application program to send back string messages whenever certain events (like button presses, etc.) occur. This way the application program determines the syntax in which Wafe talks back.

Using Wafe's Mass Transfer Mechanism

As indicated above, output lines from the application program starting with a certain prefix character are parsed and interpreted as Wafe commands. Other lines from the application are printed by Wafe to stdout. In some larger applications it is necessary to transfer a bulk of data from the application program to the frontend. In this case it is preferable to establish an additional (optional) data channel (see Figure 4), where no parsing or interpretation is performed. If an application program wants to use this data channel, it has to figure out first, on which file number Wafe is listening. The application program can obtain this information by sending the command

```
echo listening on [getChannel]
```

to Wafe which writes back for example "listening on 5". The data transferred will be stored in a Tcl variable in the frontend. If the application program issues the command

```
setCommunicationVariable \
    C 100000 \
    {sV text type string string $C}
```

the data transferred over the mass channel (5) will be stored in the Tcl variable named C. After 100000 bytes are read, the Tcl command specified in the last argument will be executed. In this example it will set the string resource of the Athena *asciiText* widget to the transferred content.

Typical Structure of Application Programs using Wafe as a Frontend

Throughout this section we assume that Wafe is used in frontend mode and an application program is performing some meaningful computations that we do not want to program in Tcl, or that we do not want to bind to Wafe. When an application program is started using Wafe as a frontend we can distinguish three phases (see also Figure 5):

1) Wafe starts the application program as a subprocess.
2) The application program creates and configures the widget tree, submits Tcl procedures and realizes the widget tree.
3) In a read loop the application program accepts commands in the form of ASCII strings from the frontend. The commands are triggered by callbacks or actions.

For some interpretative programming languages it is preferable to send an initial command from the frontend to the application process after the fork to initiate step 2. For instance in Prolog, it is convenient to send a startup goal "`[myapp], widget_tree, read_loop.`" in order to load the application "myapp" and to cause Prolog to print the commands necessary for 2 and to continue with 3. For this purpose the resource `InitCom` is provided, which can be specified in a resource file or by using the "`-xrm '*InitCom: ..`" command line option.

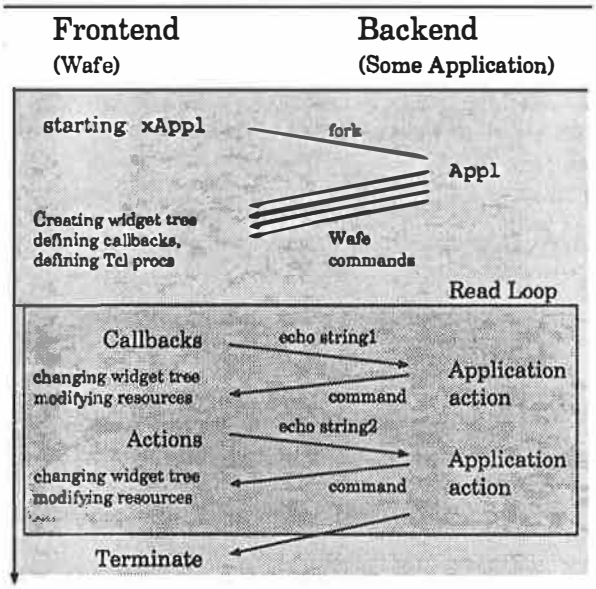


Figure 5: Using Wafe as a Frontend

The following short sample program written in Perl demonstrates steps 2 and 3. The program computes prime factors for integers typed into an Athena asciiText widget.

```
#!/usr/local/bin/perl
$|=1; # set output unbuffered

# build widget tree
print
   "%form top topLevel\n"
   ."%asciiText input top editType edit"
   ."  width 200\n"
   ."%action input override"
   ."  {<Key>Return: exec("
   ."  echo [gV input string])}\n"
   ."%label result top label {}"
   ."  width 200 fromVert input\n"
```

```
  ."%command quit top fromVert result"
  ."  callback quit\n"
  ."%label info top fromVert result"
  ."  fromHoriz quit label {}"
  ."  borderWidth 0 width 150\n"
  ."%realize\n";

# read loop
while(<STDIN>) {
  chop;
  if (/^\d+$/) {
    print
      "%sV info label thinking...\n";
    $starttime = time;
    for($d=2,@result=();$d<=$_;$d++){
      while (!($_ % $d)) {
        unshift(@result,$d);
        $_ /= $d;
      }
    }
    print "%sV result label {"
     .join('*',@result)."}\n"
     ."%sV info label {"
     .  (time-$starttime)
     .  " seconds}\n" ;
  } else {
    print "%sV info label"
      ." {invalid input}\n";
  }
}
```

Demo Applications of the Wafe Distribution

We have developed sample application programs in Perl, GAWK, Prolog, Tcl, C and Ada talking to the same Wafe binary. The following demo applications are among the programs distributed together with the Wafe sources:

- **xwafedesign:** interactive design program for Wafe applications (see Figure 6)
- **xwafeftp:** FTP frontend
- **xwafemail:** Mail user frontend with faces, using elm aliases
- **xwafenews:** NNTP based news reader, using elm aliases
- **xwafegopher:** a simple gopher frontend
- **xdirtree:** tree directory browser
- **xbm:** bitmap and pixmap viewer
- **xwafemc:** multiple choice test answering program
- **xruptimes:** rwho monitor like xnetload
- **xnetstats:** network statistics, frontend for `netstat -i <interval>`
- **xvmstats:** system statistics, frontend for `vmstat -i <interval>`
- **xiostats:** I/O statistics, frontend for `iostat -i <interval>`
- **xwafeping:** pings several machines and shows up-status
- **xwafecf:** a simple read-only card filer
- **xwafetel:** a simple read-only Oracle front-end for looking up telephone numbers
- **xwafeora:** a more elaborated Oracle frontend with updates, capable to model an entity type with distinct attribute defined subtypes, allowing multi valued attributes. The sample program supports field completion and other funky stuff. **xwafeora** is configured via a parameter block containing the sample applications "Filing Management" and "Paper Base".
- **perlwafe:** an example program calling Wafe as a subprocess of the application program (normally, it is the other way round).

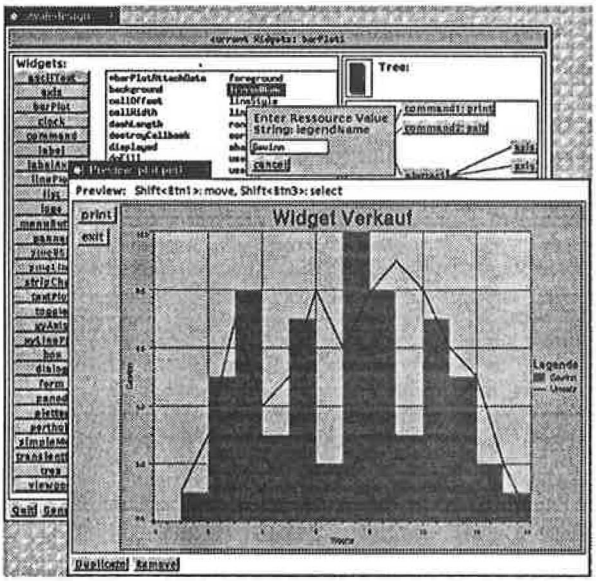


Figure 6: Sample Screen Shot of **xwafedesign** using Xaw3d and the Plotter Widget

Experiences

Our experiences proved that

- Wafe applications can be written in a wide range of programming languages,
- Wafe provides a relatively high level interface to widget applications,
- a single Wafe binary serves multiple applications,
- Wafe achieves a better refresh behavior when the application program is busy,
- click ahead is possible due to buffering in the I/O channels,
- Wafe allows better separation between user interface and application program matters,
- from its performance a user cannot distinguish whether a widget application was developed using C or Wafe,
- there is no need to program in C in order to develop widget frontends, and
- migration from existing ASCII based programs to X Window applications is easier using Wafe.

For the click ahead feature mentioned above it is questionable whether this is a desireable feature. It can be deactivated by setting widgets insensitive or by writing a small Tcl procedure which checks for each interesting callback procedure whether the program is in a busy state or not and writes accordingly friendly messages to the user.

The main disadvantage of Wafe is – when compared to widget programming in C – the higher resource consumption, because every Wafe application needs an additional process (the frontend). Frequently it is necessary to duplicate data (such as a text to be displayed in a text widget), since one copy has to be available in the frontend and another copy in the application process.

Availability

Wafe was developed on DECstations 5000/200 under Ultrix 4.2 using X11R5, and has been compiled on SparcStations under SunOS 4.1, RS6000/320 under AIX and on HP 9000/720 under hpux 8.05. Wafe can be compiled for X11R5 and X11R4. The preferred program-to-program communication is done via socketpair. Support for PIPES and SYS V streams is included for systems without the socketpair system call. The actual Wafe version and the sample applications mentioned above can be obtained via anonymous FTP from

```
ftp.wu-wien.ac.at:
pub/src/X11/wafe/*
```

(ip address: 137.208.3.4). At the time of the conference at least version 0.93 will be available. Since Wafe was announced first in May 92, about 2200 FTP-requests for Wafe were issued at the mentioned server.

References

[1] John K. Ousterhout, *Tcl: An Embeddable Command Language*, Proc. USENIX Winter Conference, January 1990.

[2] John K. Ousterhout, *An X11 Toolkit Based on the Tcl Language*, Proc. USENIX Winter Conference, January 1991.

[3] Larry Wall, Randal L. Schwartz, Frogramming Perl, O'Reilly & Associates, Sebastopol 1991.

[4] Joel McCormack, Paul Asente and Ralph Swick, *X Toolkit Intrinsics – C Language Interface*, Massachusetts Institute of Technology, 1990.

[5] Ralph Swick, Terry Weissman, *X Toolkit Athena Widgets – C Language Interface*, Massachusetts Institute of Technology, 1990.

[6] Arnaud Le Hors, *The X PixMap Format*, Part of the xpm distribution, export.lcs.mit.edu, 1991.

[7] Adrian Nye, Tim O'Reilly, *X Toolkit Intrinsics Programming Manual*, Second Edition, O'Reilly and Associates Inc., Sebastobol 1990.

[8] *X Toolkit Intrinsics Reference Manual*, Third Edition, O'Reilly and Associates Inc., Sebastobol 1992.

[9] Thomas Berlage, *OSF/Motif, Concepts and programming*, Addison-Wesley, Wokingham 1991.

[10] Kaleb Keithley, *Three-D Athena Widgets (Xaw3d)*, export.lcs.mit.edu, 1992.

[11] Roger Reynolds, *Rdd2 – Drag and Drop Library*, export.lcs.mit.edu, 1992.

[12] Peter Klingebiel, *AthenaTools Plotter Widget Set, Version 6-beta*, export.lcs.mit.edu, 1992.

[13] Doug Young, *XmGraph, A Motif Graph Widget*, iworks.ecn.uiowa.edu, 1992.

Author Information

Gustaf Neumann is Assistant Professor at the Vienna University of Economics and Business Administration, Department of Management Information Systems, in Vienna, Austria. His main research interests are centered around the intergration of heterogenous systems like the integration of different information analysis methods, the integration of various language layers (esp. in the field of logic programming and program transformation), applications of deductive databases and user interface issues. He has developed several free packages spread over the internet such as dvi2xx (a TeX dvi converter for HP LaserJets and IBM 3812 printers) and diac (conversion program for ASCII umlauts). Gustaf Neumann can be reached electronically as neumann@wu-wien.ac.at.

Stefan Nusser is writing his master's thesis at the department mentioned above. He can be reached over the network as nusser@wu-wien.ac.at.

Design and Implementation of a Multi-Threaded Xlib

Carl Schmidtmann – Consultant to Digital Equipment Corporation
Michael Tao – Xerox Corporation
Steven Watt – Consultant to Xerox Corporation

ABSTRACT

In the MIT X Window System's library Version 11 Release 5 (Xlib) there is minimal support for multi-threaded applications. Programmers writing multi-threaded programs using Xlib are required to provide locking or designate a single thread to handle many of the calls to X functions in their programs.

In this paper we will describe the design and implementation of an upgraded version of Xlib that provides more support for multi-threaded applications. Our goals were to make as few changes to the Application Programming Interface as possible, make the locking invisible to the programmer using the library, and maintain the current portability and performance of the library. This library was implemented on Digital Equipment Corporation's version of OSF/1 using the Pthreads library and Xerox Corporation's Cedar environment.

Introduction

Today, most of the graphics workstations have a single, high performance CPU. However there are several new systems available now, and more on the way, that contain several CPUs to provide higher levels of performance. To take better advantage of these multi-processor systems, most of the major workstation operating systems are beginning to include support for multi-threading. Multi-threading allows the programmer to divide an application into several parts such that the parts can run concurrently. This allows one process to use several CPUs at one time.

While multi-processor systems are the driving force behind vendor support of multi-threading, even single CPU systems can benefit from multi-threaded programming. Multi-threading provides better handling of asynchronous events and background processing within a single program. Multi-threading can also simplify writing a program that needs a responsive user interface, while at the same time performing long computational tasks or waiting for responses from slow interfaces. In general, a well written multi-threaded program will be more efficient at utilizing the available resources whether running on a single or multiple processor system.

The X Window System has become the de facto standard for high performance workstations. All major and most minor vendors of these workstations supply an X server and X libraries with their systems. Most of these implementations of X are based directly on the code supplied by the X Consortium with only minor changes incorporated for the target platform.

Currently the support for multiple threaded applications is very limited in the X Window System as it is supplied by the X Consortium. With the trend toward multi-processor systems and multi-threaded operating systems, full support for multi-threading is critical if X is to remain the standard for high performance graphics workstations.

Our goal of creating a multi-threaded version of Xlib started as part of another project to port an application from a multi-threaded, proprietary windowing system onto X. Our first pass was a "quick and dirty" patch of the X11 Release 4 (X11R4) Xlib code. After that was done, we moved on to Release 5 (X11R5) Xlib code with the intention of making our work available through the X Consortium as part of the standard releases. This paper describes our work so far and points out some of the work that should still be done.

Terminology

Process Context – A standard system process including the data address space, the program text and at least one execution thread. The process context also contains a single set of other operating system resources such as file descriptors, socket connections, etc.

Thread – A single sequential flow of control that executes within a process context. Each thread has its own local stack frame, but shares its data area and heap with all other threads within the same process context. Threads are scheduled in a fashion similar to standard process scheduling. Once a thread is scheduled to run, it starts executing and continues until it blocks (waiting for a resource to become available), or it uses up its time slice. Then,

another thread is scheduled to execute. Threads are also referred to as lightweight processes.

Single-Threaded – A program with only a single execution thread or a library not designed to be used in a multi-threaded environment.

Multi-Threaded – A program that uses multiple execution threads, or a library that allows concurrent access from multiple threads.

Thread Safe – A library that can be used in a multi-threaded program without the use of special precautions or locks by the application programmer. This does not mean that the library is multi-threaded. An example of a thread safe, but not multi-threaded, library is one that uses a locking mechanism to serialize all access to the library. In that way, no more than one thread can run within the library at a time.

Multi-Processor – A system that contains multiple CPUs. This paper only considers multiprocessing systems that have shared memory.

Inter-Thread Communication – Any one of several methods by which two threads can exchange information. Usually, this involves writing a message to an area accessible to both threads, and then notifying the receiving thread that the message is present. Whatever the method, both threads must know in advance what the semantics and mechanics of the exchange will be.

Mutex (MUTual EXclusion) – a synchronization object that controls access to a shared resource. The use of a mutex has the effect of serializing access by multiple threads to the protected resource.

Crowd Monitor – a synchronization object that permits multiple read and exclusive write access to a resource. It is usually used for a resource that is read frequently by several threads and rarely updated.

Condition Variable – a synchronization object that allows one thread to suspend its execution until another thread notifies the first thread that it can resume execution. Condition variables are usually used for producer/consumer synchronization where the consumer waits on the condition variable when it needs a resource that is not currently available. The producer then "notifies" the condition variable to awaken the consumer when the resource is available.

Deadlock – a condition in which two or more threads are waiting for a condition that will never occur. For example, each of two threads is holding a resource that the other one is waiting on. Thread 1 gets a lock for resource A and then is preempted by thread 2. Thread 2 gets a lock for resource B and then attempts to get the lock for resource A. Thread 2 is blocked because thread 1 has the lock. Thread 1 then resumes and attempts to get the lock for resource B. Now, both thread 1 and thread 2 are blocked. This deadlock cannot be resolved unless some third party kills either thread 1 or 2.

Background

Multi-threading is a paradigm that allows a single process to act as if it were several processes that all share a common environment. Multiple threads running in a single process may be scheduled independently and can be preempted by each other.[1] The reason for multiple threads in a single process context, instead of multiple processes using shared memory, is that the overhead of thread switching is significantly lower than that of process switching. In addition to sharing data, threads also share file handles, socket connections, and other per-process resources.

For this paper we are using the mechanisms defined by the IEEE POSIX 1003.4a draft proposal. Other multi-threaded systems will provide similar functionality but may use different terms.

Using a Single Threaded Library

There are several possible methods of making a library thread safe [Jon91].

Designated Thread

One method to make a library thread safe is to use the library in a single-threaded manner. To use a library that is not thread safe in a multi-threaded program, the application programmer designates one thread in the program as the interface to the library. All other threads needing access to the library must use inter-thread communication with the designated thread, which will make the call and return any results to the originating thread by way of inter-thread communication. This method is useful if the source code for the library is not available, but it puts the burden on the application programmer. This extra effort can easily amount to more than the effort saved by using the library.

Encapsulation

Another way to make a library thread safe is to put a wrapper around the library so all calls into the library go through the wrapper. The wrapper functions lock a mutex and make the library call. This method makes the library thread safe without making any internal modifications to the library. The drawback to this approach is that the library is still single threaded, and the multi-threaded program that uses it is constrained to have only a single thread executing in the library at any one time. This may cause a bottleneck if the library is called frequently from multiple threads.

Multi-threading

The next method of making a library thread safe is to redesign and recode the library so it supports concurrent access. This will usually require a

[1] Some multi-threaded systems do not do preemptive scheduling among threads. However, software that is designed for preemption will operate properly on a non-preemptive system.

significant amount of redesign work, since most of the library functions have probably been designed and optimized to take advantage of the single threaded environment. It also requires the person modifying the library to perform a lot of reverse engineering to determine what needs to be locked, where deadlocks might occur, and which changes might impact the functionality. During redesign, the library interface may also need alterations. The changes may be required to maintain functionality, add functionality, or limit functionality.

The advantage of using a designated thread or encapsulating the library is that it does not require any modifications to the library. These are the only alternatives if the source code for the library is not available. The disadvantage of these two methods is that the library becomes a single shared resource, and could become a bottleneck if the library is used heavily by more than a single thread. The designated thread method also places a burden on the application programmer to deal with multi-threaded issues.

We chose to modify Xlib to make it multi-threaded because the library would be heavily used by multiple threads. Also, since Xlib is usually a core library that comes with the operating system, users would expect it to be thread safe and not require any special locking or access restrictions.

The best method for making a library thread safe is to design it to be multi-threaded from the start. This obviously isn't feasible for existing libraries, but it should be considered for new work if there is any chance the new library will be used in a multi-threaded environment. It is our belief that this applies to any library now being written.

Multi-threading in R5 Xlib

Xlib has a few multi-threaded features; the display lock, the static lock in XOpenDisplay, and the event queue used in conjunction with the Display lock.

All server requests and all access to the event queue are locked with the Display lock. This protects the request buffer, the event queue, and the server connection. Presumably, this lock was meant to protect all access to the Display structure, but the current implementation allows many accesses without regard to the lock. A new patch to the header file is now hiding the internals of the Display structure from application programs, but several of the toolkit libraries violate this intent and directly access the Display structure and its resources.

The static lock in XOpenDisplay protects the XOpenDisplay function so that no more than one thread can be executing the function at a time. This keeps the static list of Display structures from being accessed concurrently by multiple threads.

The event queue allows events to be read from the server connection while waiting for a reply to a server request. It also allows the connection to be read more efficiently, by reading all events present before processing them. Access to this queue is protected by the lock in the Display structure.

Limitations in R5 Xlib

In our environment, the multi-threaded features in the R5 Xlib fell short of providing a truly thread safe library. The areas in which we found problems were the handling of the display connection, unprotected static data areas and the error handler mechanism. The concept of a single lock for all Display-related activity does provide thread safe access, but it does not allow for any parallelism. It also requires that the application programmer has knowledge of the internals of Xlib and makes allowances for it.

Server Connection

In the current implementation, if a thread is waiting for events from the server, no requests can be sent to the server until at least one event is received and queued. Then, the thread leaves the library, allowing the requests to be sent. A multi-threaded application will typically have one thread that does event dispatching and other threads that handle background tasks. If one of the background threads needs to send a request, it will almost always be blocked until an event arrives. If the user is waiting for a background process to complete and notify him by updating the display, there is a deadlock with the user.

On the other hand, if a thread sends the server a request that requires a reply, all access to the request buffer or the event queue is blocked until the reply is received and the thread returns.

Static Data

Another problem with the R5 Xlib in a multi-threaded environment is that several sections of the code depend on static, and sometimes global, data. Currently the only static areas that are protected by locks are the list of Displays and the event structure free list. Some examples of unprotected static data are the Quark table, the Xrm databases, the Context table, the Error Text table, the Xcms color map tables and the error handlers.

Error Handlers

The error handlers in Xlib are global function pointers that can be changed at any time by an application program or an internal function. This means that when a thread registers an error handler, there is no guarantee that the handler will still be installed when the error it was supposed to handle arrives. There are also internal routines that install their own error handlers, make server requests, and then restore the original error handler. This can cause unpredictable behavior. For example, one thread saves the

current error handler, installs a new error handler, and then waits for a response from the server. While the first thread is waiting, a second thread installs a new error handler. The consequences of this case are that the first thread's error handler is not active when it is needed, and when the first thread restores its saved error handler, the second thread's handler is lost.

Goals of This Implementation

Our work to create a thread safe Xlib was part of a project to port Xerox's GlobalView product to X. GlobalView is a large multi-threaded office automation package which was using its own windowing system. Because of the project deadline it was vital that the first pass at the library be completed quickly. Therefore, all the first pass accomplished was to make event handling and server requests thread safe. This work was done on X11R4.

During this stage of the project, X11R5 was released, so it was decided that working on the latest version of the code would be more productive than having to do everything twice. It was also decided at this time that our work would be contributed to the X Consortium. The goals of this second effort included all of Xlib and were better defined than the first pass.

Robust Multi-Threading

The goal here was to make the library truly multi-threaded and not just thread safe. The library needed to allow for parallelism where possible (e.g., read events and send requests simultaneously) without the application needing to know what calls were safe to execute in parallel. It also meant that thread support would be built into the code without kludges.

Same Code For Multiple and Single Threads

All the changes that were made had to support both multi-threaded and single threaded applications. This goes beyond putting `#ifdef`'s around everything. The library should be compiled with multi-threaded support if it is available on the platform for use in multi-threaded applications. A single threaded program should be able to use the same library file (shared or statically linked) without any knowledge or special code for multi-threading.

Performance

The performance of the new Xlib should be equivalent to the X11R5 Xlib when running a single-threaded application. For multi-threaded applications the performance should be better than could be obtained by using a designated thread for all Xlib calls.

No Interface or Functionality Changes

While this was a goal, we knew from the start that there might be some things that couldn't be made thread safe. We also knew there would probably be some extra functionality that would be useful in a multi-threaded environment. The strongest incentive to keep to this goal was that the only user documentation we had to write was to describe any differences in the interface.

Minimal Structural Changes

While the internals of a library can be changed without affecting application programs, we tried not to modify the internal workings unnecessarily. The reasons for this were a need for our work to be accepted by the Consortium and because the less we changed the less we might break.

Library Modifications

The modifications made to Xlib for the project fell into three categories: general modifications to ensure appropriate resource locking, enhancements to existing functionality, and introduction of new functionality.

General Modifications

The problems involved with the Xcms, Quark and Xrm routines were the same: The need to lock static data against multiple initialization and multiple writers. The code was analyzed to determine the best places a single lock could be added simply. In most cases, these functions are of a fast-in-fast-out nature, so there is not much probability of long waits on the lock. The exceptions to this generalization are the various routines that do operations on entire databases.

A lock has been added to the Xrm database structure to serialize manipulations of the database. However, there are still some potential sequencing problems; if one thread stores one value for a resource, and another thread stores another value at (almost) the same time, it is simply a race to see which thread's data actually gets stored, and which one's data falls on the floor. Also, with the current scheme, only one thread may be searching the database at a time.

Fortunately, writes to these databases are generally done only at initialization time, and most accesses are read-only after that. There are specific enhancements that could be applied to both Xcms and Xrm, but resources were not available to investigate them fully. They are described in the Future Work section.

Enhanced and New Functionality

We have added some new features and modified some of the existing functionality. These changes required some additions to the Xlib interface but we have tried in all cases to leave the original interface and functionality intact. We have modified the internal access to the server connection so that a thread waiting to read from the connection does not prevent a thread from writing to the connection. We have also allowed for round trip

requests to be handled while there is a thread waiting for events. For multi-threaded programs that divide the handling of events among several different threads, we have implemented support for multiple event queues for a single server connection. Along with the multiple event queues, we have added a timeout option when requesting events. This means that a thread doesn't have to block forever waiting for an event. An area in which we both enhanced and added functionality was the error handling mechanisms.

Server Communications

A mechanism was added to control reading the server connection without blocking access to any other part of Xlib. This allows an event handling thread to spend most its time waiting for events without preventing other threads from sending requests to the server.

To implement this, it was necessary to modify the round trip request handling by creating a queue for replies. Now, access to both events and replies is done by way of queues. A thread sending a round trip request will check the reply queue, and a thread waiting for events will check the event queue. Each queue has a mutex that controls access to the queue. After locking the mutex, the thread checks to see if what it is looking for is already available. If a reply or an event is available, it is taken off the queue, the mutex is unlocked, and the reply/event is returned. If the queue doesn't contain what the thread is looking for, the thread will attempt to read from the server connection. Refer to Figures 1 and 2.

All read access to the server connection has been consolidated into a single routine, access to which is controlled by a semaphore-like mechanism. Whenever a thread needs to read from the server connection, it locks the read mutex and checks the read flag. If the flag is set, the thread locks the appropriate queue mutex, unlocks the read mutex, and waits on the queue condition variable. If the read flag is not set, the thread sets it, and unlocks the read mutex. The thread then reads from the connection. Note that the read flag controls access to the connection, and the read mutex controls access to the read flag.

When an event or reply is received, the appropriate queue's mutex is locked and the event/reply is placed on the queue. The queue's condition variable is then notified to wake up any thread waiting for that queue. The queue mutex is then unlocked. If the thread doing the reads is not interested in what was received, it continues to read from the connection.

If the thread was interested in the reply/event received, it will stop reading from the connection. The reading thread then locks the read mutex, clears the read flag, and wakes up a thread waiting on a queue condition variable, so the waiting thread can start reading. The thread that was reading from the server connection can then get the event/reply from the queue and return it. A thread waiting on a queue condition variable must check the queue when it wakes up to see if what it was waiting for has arrived. If the anticipated event has not arrived, the

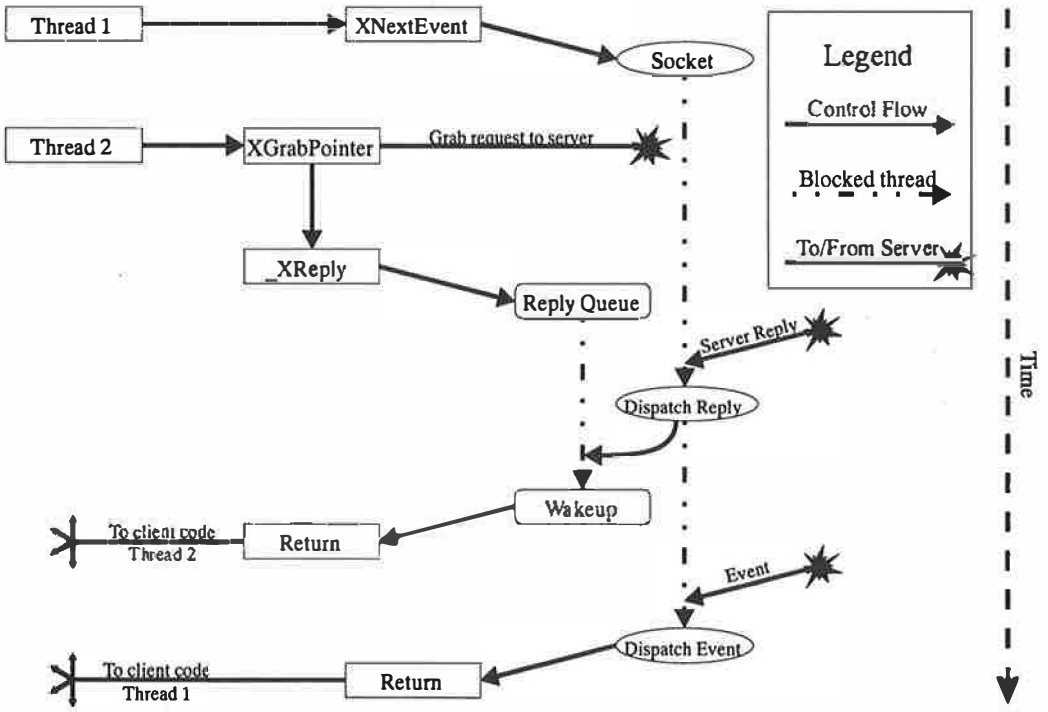


Figure 1: Round trip request while waiting for events

thread tries again to gain read access to the connection.

A consequence of having a reply queue is that it is possible to have more than one outstanding round trip request. The server is still going to process the requests serially, so this is not a large performance enhancement, but it allows an overlap between transmission time and request processing.

Hooks have been added to the code to allow implementation of separate read and write threads. This would allow a thread to be dedicated to reading from the server connection and putting whatever arrives onto a queue. Another thread could be designated to perform all writing to the connection. These extra threads would save some checking and locking overhead but cause a little more thread switching. The mechanism has not been implemented yet because it requires changes and/or additions to the Xlib interface to allow the application to indicate the desire to start these extra threads.

Multiple Event Queues

The new multiple event queuing design is closely fitted into the original Xlib architecture. All existing Xlib event retrieving functions operate on the default event queue. A new set of multiple event queuing functions operate on any created event queues as well as the default event queues. Multi-threaded clients can create multiple event queues and select any events to be dispatched onto these event queues. The same event can be placed on multiple queues. A mask and a predicate function are used together to filter the events placed on a queue. Each event received from the server will be checked against the mask for the queue. If there is a match and the predicate function is non-NULL, it will be called. If the predicate function returns true or is NULL, the event is added to the queue. If the mask doesn't match or the predicate returns FALSE, the event will not be put on the queue. Each event is checked against all queues so that a single event can be placed on more than one queue. For performance reasons, the number of queues should be kept to a minimum. Also, masks should be used to limit the number of times a predicate function is called, and predicate functions should be as fast and short as possible.

When the client opens a display connection, a default event queue is created with default values of all 1's for the mask, and null for the predicate. The

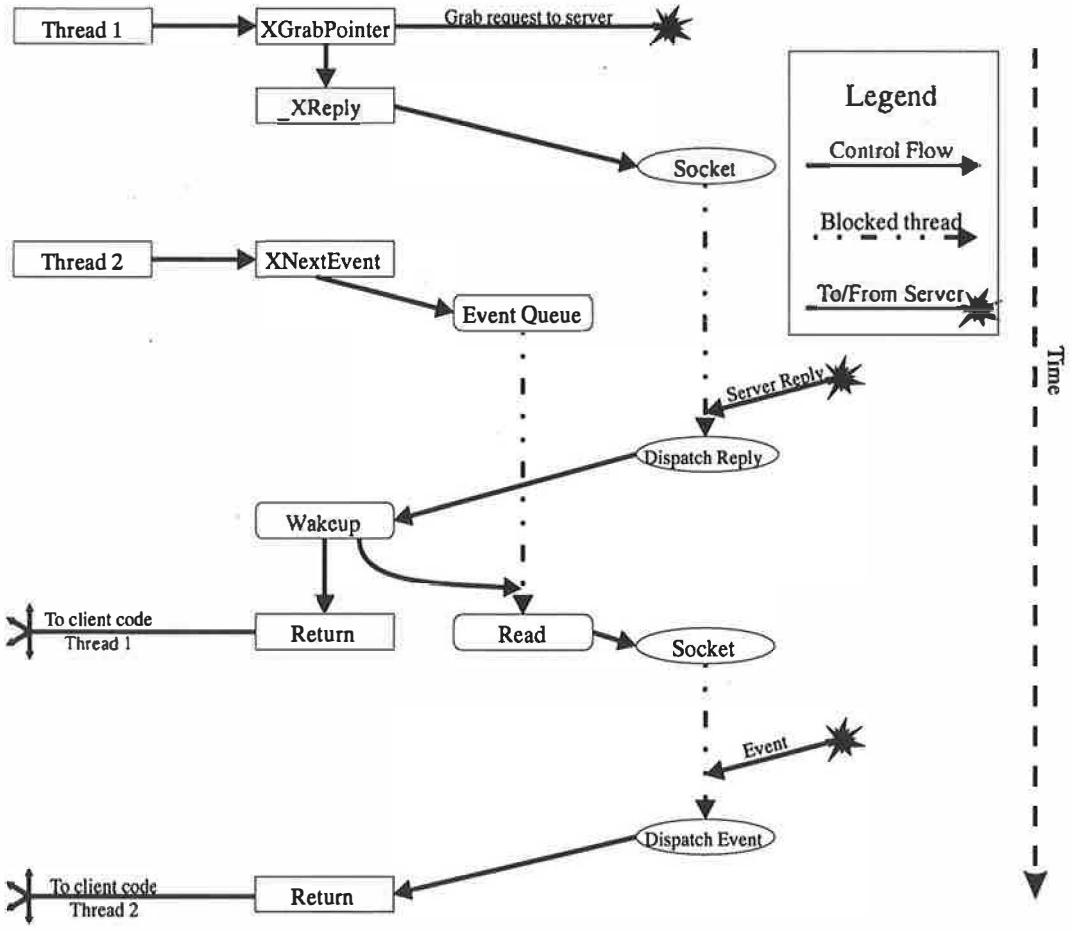


Figure 2: Interleaving round trip request and events

original Xlib event functions will operate on this event queue.

Each event queue is designed to be accessed by a single thread only. If a client chooses to access an event queue in multiple threads, the client must guarantee the mutual exclusion of accesses to the event queue.[2]

Timeout Support for Event Retrieving Functions

We implemented a new version of each of the existing event-retrieving functions that block. These functions have a timeout parameter, and return TRUE if an expected event is received within the timeout interval. Otherwise, they will wait until the timeout interval has expired, and then return FALSE.

After studying the behavior of the Xerox multi-threaded application GlobalView for X (GVX), we noted some bottlenecks arising out of the use of some of the new Xlib features.

GVX uses several threads to handle event dispatching. Each of these threads has an event queue and most of them use the timeout feature while reading events from the queue. In our first implementation an XFlush call was made every time an event queue was searched for an appropriate event. This resulted in many small output buffer writes. This behavior was changed by calling XFlush only once per call to an Xlib event retrieving function. If events arrive but are not of interest to that thread, it does not call XFlush again. This cut the number of buffer writes by 50%.

Multiple Output Buffers

Another thing we noted was that GVX did a lot of image transfers and this tended to cause the buffer to fill up quickly and then blocked the thread doing the drawing while the buffer was being written. To alleviate this bottleneck multiple output buffers were implemented. This allowed one thread to be writing a full buffer to the server while other threads continued to store requests into another output buffer. To make this more efficient for the drawing threads, another thread was created to handle writing of output buffers in the background.[3]

The performance improvements were mixed when using multiple output buffers and a background writer thread on a single CPU system.[4] No improvement in performance was noted when the application was generating an image and trying to display it. However, a 10-20% improvement was noted when the application was retrieving information from a file and displaying it. The improvement was most likely due to the overlapping of I/O processing.

Error Handling

The global error handlers, _XErrorFunction and _XIOErrorFunction, were replaced by three per Display handler lists. There is a non-fatal handler list for both internal and external (to Xlib) error handler functions. There is also a list for fatal error handlers. When a non-fatal error occurs, usually an unexpected error message from the server, the thread reading from the server connection calls each handler on the internal non-fatal error handler list. If one of them handles the error, it returns TRUE and no more handlers are called. If none of the internal handlers returns TRUE, then each of the handlers on the external list is called. If one of these handles the error, it returns TRUE and no more handlers are called. If no handler returns TRUE, the default non-fatal error handler is called.

Fatal errors are handled in a similar fashion. However, the return value of the handlers is ignored and all the fatal error handlers are called. After the handlers are called, the default fatal error handler is called.

New functions were added to the Xlib interface to allow application programs to add and remove handlers from the handler lists. Error handlers are not guaranteed to be called within the thread that registered them. They are also not guaranteed to be called in any particular order.

This new way of dealing with error handlers allows applications to have an error handler watching for its errors, and to have cleanup routines registered if there is a fatal error. By making the error handlers a per display resource it is now possible to have a display connection crash without the entire application crashing with it.

To support existing applications, the functions XSetErrorHandler and XSetIOErrorHandler each register a handler as the default non-fatal and fatal error handlers respectively.

Results

After releasing the first version of the multi-threaded Xlib we reviewed our original goals and tried to determine how close we came to meeting them. This section describes the results for each of our goals.

Robust Multi-Threading

We believe we have made Xlib as multi-threaded as the current interface will allow. In some cases it was not possible to make the locking completely invisible to the application. There were interface functions that allowed the application to do direct get and set operations on elements of the Display structure (i.e., XrmGetDatabase,

[2]This is arguably a bug, and could be fixed with another layer of locking. However, accessing one queue from multiple threads will probably produce unpredictable results.

[3]This implementation was only a test using the Cedar threads package. We have not yet built a portable method for creating this extra thread.

[4]Sun SPARC 2 running SunOS 4.1.1 and Xerox's Cedar environment.

XrmSetDatabase). We did make it possible to have separate threads handling events and sending requests without external synchronization.

Same Code For Multiple and Single Threads

We have run standard single-threaded applications on top of the multi-threaded Xlib compiled both with and without multi-thread support. Not all of the applications worked with the multi-thread support enabled, most notably twm. We have not been able to debug this yet.

Performance

For performance testing we used a DEC DS3100 with 24MB of RAM and an 8-bit color display and a DEC DS5100 with 24MB of RAM and no display. Both of the systems were running DEC OSF/1 V1.0 which is a developer release. The server was the DEC Xws server that comes with the OS. The network was thin wire ethernet and included only these two systems. The single machine tests were also run on a MIPS Magnum 3020 running LynxOS and an X11R5 server.

For event handling tests we used a simple program that timed how long it took to read events using XNextEvent in a loop. Another program that just sent client message events was used to generate events. To measure the entire event handling time including reading the socket, the sending program sent one event, waited for 2 seconds, and then sent a large number of events. The receiving program would receive the first event, wait 3 seconds, and then enter the timing loop. This was to insure that events were waiting in the socket but not yet read into the event queue.

The request testing used a program that timed executing XGetInputFocus in a loop. The event reading and request testing programs were each compiled and linked with the R5 Xlib and with the multi-threaded Xlib compiled with multi-threading support turned on. The event generating program was only linked with the R5 Xlib. The tests labeled local were run with all programs, including the server, executing on a single machine. The remote tests were run with the event reading and request testing programs executing on a different machine from the server. The event generating program was always executed on the same system as the server.

In all tests the multi-threaded Xlib was slower than the R5 Xlib. This is due, in part, to the added overhead of the locking calls. There was also some overhead added because of the extra queue for replies and the support for multiple event queues. The results of these tests are summarized in Table 1.

There was quite a large difference in the performance of the multi-threaded Xlib in the local tests versus the remote tests. In the local tests, the multi-threaded Xlib took 3 to 4 times as long to read events from the queue as the R5 Xlib. Running remotely from the server and the event sending program, the multi-threaded Xlib took only 1.1 to 1.5 times as long as the R5 Xlib. We have not determined the cause of this disparity.

For the request handling, the times were 1.2 to 1.5 times longer for the multi-threaded version of Xlib. There was not a dramatic difference between the local and remote tests.

No Interface or Functionality Changes

We have added functionality with this version of Xlib but all of the R5 functionality has been left intact. Use of some functions is strongly discouraged in multi-threaded programs but they are still available.

Minimal Structural Changes

Most of the structural changes made were restricted to the event and request handling. Many new locks were added to try and relieve some of the contentions for the DisplayLock which was used for all the locking in the R5 version of Xlib. The handling of access to the server connection was completely changed to localize and control this access. A reply queue, as well as support for multiple event queues, was added. The error handling was also changed to add support for multiple error handlers.

Future Work

Many tasks need to be completed to make X truly multi-threaded, most of which are design and structural changes. Our approach was to do as much

Event Handling Times	R5 Local (ms)	MT Local (ms)	R5 Remote (ms)	MT Remote (ms)
1000 events	324	987	360	561
10000 events	4193	16049	4301	4864
100000 events	44981	173437	43999	48737
1000 requests	4933	7415	4232	4976
10000 requests	49099	73684	42094	49459
100000 requests	490806	721585	420772	494651

Table 1: Event and request handling performance

of the job as was needed by our clients, and whatever else we had time for. This forced some tasks to be put on hold.

Xlib Enhancements

Extensions

A major area that still requires work is the support code for the extension library. This code provides support for most of the X extensions (e.g., MIT-SHM, PEX, SHAPE). This code cannot be made thread safe without doing major redesign work. Most of the extension skeleton code does not take multi-threaded issues into account, and none of the standard extensions that we examined addressed multi-threaded issues. The redesign of Xlib internals also forces a change in the way that extensions are written. Each standard extension should be examined and redesigned with multi-threaded issues in mind. These changes must reflect the different way that reply and error handling work, and locking must be carefully considered. We did implement the MIT-SHM (shared memory) extension by writing a new interface that handles locking and calls the MIT routines, but this implementation is far from optimal.

Possible Restructure of Xrm Functions

Xrm has one major obstruction to multi-threading: The XrmGetDatabase and XrmSetDatabase routines. These routines manipulate the per-display database pointer. Once a thread has that pointer, it can do whatever it likes to the database, and then reset the per-display pointer. If two threads do this, there is no predicting what will happen. For the current implementation, it was decided that this is a client problem; don't use XrmGetDatabase or XrmSetDatabase once multiple threads are running. Unfortunately, that violates one of our primary goals, which was to hide all locking from the application programmer.

Xrm also needs more work with locking, primarily to take advantage of crowd monitors. This would allow multiple threads to read one database at the same time, while enforcing correct behavior during writing. The current implementation allows only a single thread to be reading from a given database at one time.

Possible Restructure of Xcms Functions

The Xcms system is very heavily self-referential code, and was very difficult to make thread-safe. It makes calls to other modules of the library, which can call back into the Xcms module. This breaks the lock hierarchy necessary to prevent deadlocks. This code could also benefit from the use of crowd monitors to allow multiple readers.

Redesign Xcms Colormap Search

The function XcmsCCCOfColormap and the function it calls, CmapRecForColormap, need to be looked at very carefully. XcmsCCCOfColormap currently runs through a loop for every visual of every screen on the display. It creates a window with the passed-in Colormap on each visual, and waits for the BadMatch to come back from the server. This process is fine in a single-threaded system, but since it calls XCreateWindow, it will not work in a multi-threaded system.[5] The quick fix would be to move the XCreateWindow request formatting into CmapRecForColormap, but it would be preferable to come up with a better implementation. Two possibilities are: adding a new protocol request, or using a general resource identification extension.

Separate Reading & Writing Threads

Another area that needs further study would be to create one thread that does nothing but handle reading the server connection and another thread that only does writes to the server. This would mean that a thread sending requests would never have to incur the overhead of writing the request buffer to the server. It would always just enqueue its request and continue on. This approach will probably show the most improvement on a multi-processor system.

There is some support for separate reader and writer threads in this version of the library. However, we have not provided a generic method to allow either of these threads to be created. A proper implementation of a method for creating these threads would require a modification to the Xlib interface.

Internationalization

The Xsi internationalization support in X11R5 gives some indication that it was designed with multi-threaded issues in mind. The Ximp support does not. The resources to fully investigate the internationalization support in X11R5 were not available, and our client had an existing, proprietary method for handling multi-national text.

Related Work

Connection Status Call

A call should be added to Xlib to allow toolkits to avoid networking-system dependencies. This call would allow a toolkit or client program to ask if input is available on a list of display connections, possibly giving a timeout argument. This call is not strictly required in a multi-threaded system, but could be a useful alternative to using XConnectionNumber on several display connections and having to call the poll or select routines directly. This would improve portability of multi-display applications, since they would no longer have to know the correct system interface for multiplexed input. This call would be better than the retrieve event with

[5] This method for finding a visual for a Colormap causes a deadlock because it calls XCreateWindow with the Display lock held. XCreateWindow also locks the Display lock, resulting in a deadlock on some systems. This method must be used synchronously, causing potentially long delays.

timeout functions that we added, if non-blocking event reading methods were used.

Multi-threaded X Toolkits

Sun has done a thread safe version of Xt and XView [Sma92]. However, these versions treat the entire Xt and XView subsystems as locked resources, such that only one thread can be running in any part of the libraries at a time. This greatly reduces the concurrency possibilities in a heavily graphics-oriented application, or in one that has a large number of threads that use the toolkit.

The need for higher-level libraries to be multi-threaded is becoming critical. Currently, OSF/1 has threads, as do Solaris 2, VMS, AIX 3.2, System 5 Release 4, and SunOS 4 with the "lwp" library. Each of the higher-level toolkits and widget sets for these platforms should be written to allow multiple threads to do useful work at the same time. A toolkit and widget set that are multi-threaded should be made available as part of the core X distribution.

Multi-threaded X Applications

Applications will need to be written to take advantage of the multi-threaded model of computing; multi-threaded libraries are not useful if there are no multi-threaded clients to use them. It is unlikely that this will be a problem: there is already one major software package using this code, and more are in development.

In a multi-threaded environment, it will be possible for a program to have one thread doing data acquisition in the background, displaying its data as it receives it. Another thread could be doing computations on the received data, and displaying the results of the computations. A third thread would handle user requests and events. This resulting application would have better responsiveness and cleaner code structure than one using the callbacks, timeouts, and work procedures employed with current toolkits.

Conclusions

The Consortium must create a policy for multi-threading.

In the authors' opinions, it is imperative that the X Consortium require new additions to the core X distribution to be thread-safe. If this commitment is not made, then the state of the library will oscillate from fully multi-threaded to its current inconsistent state, where some things work, some things might work, and some things cannot work.

Multi-threaded Xlib is possible.

We believe we have demonstrated that the current Xlib code from the X Consortium can be made multi-threaded without breaking the interface. The library we have created works with current single-threaded applications without modification. There will undoubtedly be some applications that break because they were cheating and calling internal Xlib routines that were not part of the external interface. Toolkits are the biggest violators of the Xlib interface but as we pointed out, they will need work to make them multi-threaded also.

Multi-threaded Xlib is needed.

The need for a multi-threaded Xlib (as well as other libraries, like Xt, Motif, OLIT, etc.) is clear from the fact that most major workstation suppliers are now shipping a multi-threaded operating system with their workstations. If they are not shipping thread safe libraries with those systems, the application developers are going to be in for some unpleasant surprises.

There is still much work to be done.

The work to be done to supply fully multi-threaded libraries for X is significant. Xlib still needs some work, most of the toolkits have not even been examined with multi-threading in mind, and from what we can tell nobody has done anything with the widget sets. Since some of the widget sets are built and maintained by suppliers of multi-threaded operating systems, we assume they are working on it. Finishing this work will require some cooperation from the groups that have built the different portions of the libraries, the major workstation vendors, and the X Consortium.

New X features must be designed with multi-threading in mind.

Lastly, we strongly urge anyone who is working on updates or additions to the libraries make their code multi-threaded. We feel it would be a mistake for the Consortium to take on the task of supporting two versions (single threaded and multi-threaded) of each of the libraries. This would make maintenance a nightmare. However, it would be as big a mistake to leave the code as it is and let the suppliers of the multi-threaded systems each make their own updates to create a multi-threaded library: that invalidates any attempt at standardization.

Acknowledgments and Thanks to:

Xerox Corporation, and Digital Equipment Corporation, cosponsors of this effort to produce a multi-threaded Xlib.

Ann Ting (Xerox), Brian Hoshiko (Xerox), and Alex Phillips (Digital), without whose support the multi-threaded Xlib project would not have been possible.

Rita Laughter, for editing and driving our writing effort (we needed it).

Catherine Watt, for final grammatical and structural editing magic (we really needed it).

References

[Atk89] Atkinson, Demers, Hauser, Jacobi, Kessler, Weiser. "Experiences Creating a Portable Cedar" SIGPLAN '89 Conference on

Programming Language Design and Implementation. June 19-23, 1989.

[Bir89] Andrew D. Birrell, "An Introduction to Programming with Threads." Research Report #35, January 1989. Digital Equipment Corp. Systems Research Center.

[Jon91] Jones, Michael B., "Bringing the C Libraries with Us into a Multi-Threaded Future." USENIX Winter 1991.

[Ora91] O'Reilly & Associates, Inc. "Guide to OSF/1: A Technical Synopsis" ch. 2, 11.

[Sma92] Bart Smaalders, Brian Warkentine and Kevin Clarke, "Prototyping MT-Safe Xt and XView Libraries." The X Resource, Issue 1, pp. 91-107 Winter, 1992.

[Smi92] John Allen Smith, "The Multi-Threaded X Server." The X Resource, Issue 1, pp. 73-89, Winter, 1992.

[Wei89] Weiser, Demers, Hauser. "The Portable Common Runtime Approach to Interoperability" ACM Symposium on Operating Systems Principles. December, 1989.

Author Information

Carl Schmidtmann is a principal of Faultline Software Group, Inc. Lately he has been consulting in OSF1 and implementing a multi-threaded version of MIT's Xlib. He has been working with the X Window System and Motif for the past four years. He also teaches a C programming course and a X/Motif programming course. He can be reached at cws@faultline.com.

Michael Tao worked at Xerox Corporation for almost seven years. During that time, he designed and implemented various multi-threaded system libraries as parts of Xerox's network filing system and GUI toolkits. For one and a half years, Michael was the project leader in the effort to develop multi-threaded X Window System libraries. Michael now works for Sun Microsystems on SVR4 system libraries. His current electronic mail address is tao@eng.sun.com .

Steven Watt now works for Lynx Real-Time Systems, but used to be a consultant in the San Jose area for 7 years. His previous clients include IBM and Xerox, as well as several smaller companies. His primary focus has been on the X Window System, doing client and library development for various platforms. His interests include embedded and multi-processor systems and communications systems. He can be reached via electronic mail at steve@wattres.sj.ca.us .

The Design and Implementation of the Inversion File System

Michael A. Olson – University of California at Berkeley[1]

ABSTRACT

This paper describes the design, implementation, and performance of the Inversion file system. Inversion provides a rich set of services to file system users, and manages a large tertiary data store. Inversion is built on top of the POSTGRES database system, and takes advantage of low-level DBMS services to provide transaction protection, fine-grained time travel, and fast crash recovery for user files and file system metadata. Inversion gets between 30% and 80% of the throughput of ULTRIX NFS backed by a non-volatile RAM cache. In addition, Inversion allows users to provide code for execution directly in the file system manager, yielding performance as much as seven times better than that of ULTRIX NFS.

Introduction

Conventional file systems handle naming and layout of chunks of user data. Users may move around in the file system's namespace, and may typically examine a small set of attributes of any given chunk of data. Most file systems guarantee some degree of consistency of user data. These observations make it possible to categorize conventional file systems as rudimentary database systems.

Conventional database systems, on the other hand, allow users to define objects with new attributes, and to query these attributes easily. Consistency guarantees are typically much stronger than in file systems. Database systems frequently use an underlying file system to store user data. Virtually no commercially-available database system exports a file system interface.

This paper describes the design and implementation of a file system built on top of a database system. This file system, called "Inversion" because the conventional roles of the file system and database system are inverted, runs on top of POSTGRES [MOSH92] version 4.0.1. It supports file storage on any device managed by POSTGRES, and provides useful services not found in many conventional file systems.

The Inversion file system provides transactions and fine-grained time travel to users by taking advantage of the POSTGRES no-overwrite storage manager. File data are stored in the database, so that file updates are transaction-protected. In addition, the user may ask to see the state of the file system at any time in the past. All transactions that had committed as of that time will be visible, so the file system state will be exactly the same as it was at that moment. This is an important improvement on the coarse-grained time travel provided by other systems.

Another feature provided by Inversion is fast recovery. No file system consistency checker needs to run on the Inversion file system after a crash, since recovery is managed by the POSTGRES storage manager. File system recovery is essentially instantaneous. Any updates that were in progress at the time of the crash, but had not committed, will be rolled back. Any committed updates are guaranteed to be persistent across crashes.

In addition, files in the Inversion file system may be located on any device managed by POSTGRES. The Inversion namespace is uniform across devices. This means that the Inversion file system can span multiple devices (and device types) transparently. For example, the current system manages data stored on a 327Gbyte Sony optical disk WORM jukebox, and on magnetic disk. In the near future, a 9 TByte Metrum VHS-form factor tape jukebox will also be supported.

Finally, the fact that Inversion is built on top of POSTGRES makes it possible to issue *ad hoc* queries on the file system metadata, or even file data itself. Instead of mastering the use of many different programs, the user may examine the file system's structure and contents by formulating simple POSTQUEL queries. In addition, indices may be defined to make file system operations run faster, at the user's discretion.

The system described here currently supports a group of physical scientists researching global

[1]This research was sponsored by the University of California and Digital Equipment Corporation under Digital's flagship research project "Sequoia 2000: Large Capacity Object Servers to Support Global Change Research." Other industrial and government partners include the California Department of Water Resources, United States Geological Survey, MCI, ESL, Hewlett Packard, RSI, SAIC, PictureTel, Metrum Information Storage, and Hughes Aircraft Corporation. Additional funding was provided by the Army Research Office under grant number DAAL03-91-0183.

climatic change as part of the Sequoia 2000 research project [STON91]. For this user community, transaction protection and fine-grained time travel are important services. The amount of storage managed requires novel fast recovery techniques like those provided by Inversion.

The rest of this paper is organized as follows. First, related work in file systems and in database management systems is presented. Next, the architecture of the POSTGRES database system is summarized, with attention to the features used by Inversion. Then the design and implementation of Inversion are described. A discussion of user-level access to Inversion files follows that. Next, Inversion's performance is measured on a benchmark based on the access patterns of its primary users. Finally, the conclusion summarizes the major points of the paper, and instructions on obtaining the code are given.

Related Work

File systems researchers have lately concentrated on providing new services to administrators and to users. Important areas of research include transaction protection, viewing past states of the file system ("time travel"), and attribute-based naming strategies.

QuickSilver [CABR88] is an early example of a file system that allows users to protect file changes with transactions.

The Wisconsin Storage System (WiSS) [CHOU85] was an early implementation of a storage manager supporting access to large data objects. WiSS decomposes large objects into pages, changes to which are protected by transaction boundaries. The WiSS client controls physical layout of object pages, making it easy to implement clustering strategies appropriate to particular large object applications. Indices on logical page locations make object traversal fast.

The EXODUS storage manager [CARE86] provides a set of low-level abstractions for managing large data objects. It supports efficient versioning of these objects. Users can extend the system to support new object types and operations on them.

Episode [CHUT92] embeds transaction protection in the file system directly for file system metadata changes. These transactions permit faster recovery after a crash than do graph-traversal programs like *fsck*(8). The file system manages a write-ahead log of directory updates, and can detect and remove transaction-inconsistent states quickly. However, Episode does not provide transaction protection to users, so user files may be left inconsistent by a system crash.

Log-structured file systems [ROSE91, SELT93] append file system changes to the end of a log on disk. A special "cleaner" process periodically reorganizes storage to recover space occupied by obsolete data. [SELT90] proposes extending such systems with support for transactions, and support for time travel could be added with appropriate changes to the cleaner process.

Finally, several libraries and toolkits have recently appeared that offer transactional and other services to users. 4.4BSD includes a database access method library, **db**(3), which provides keyed access to user data [SELT92]. This library includes support for transactions, allowing users to make consistent changes to files managed by the library. Kala [SIMM92] offers primitives allowing users to implement tailored transaction management and version control systems on a persistent programming language data store.

As very large storage devices, such as optical disk and tape jukeboxes, become widely available, many researchers are investigating ways of saving historical file system states automatically. Users can then travel in time over the file system, viewing old file contents at will.

The Plan 9 file system [QUIN91] periodically snapshots file system contents to an optical disk jukebox. Only changed files need to be copied; the system automatically reconstructs the complete state of the file system from the set of snapshots on the jukebox. The granularity with which snapshots are taken is configurable, but is currently once a day.

3DFS [ROOM92] uses a similar snapshot strategy to capture and recover file system state. This system extends the file system's namespace to include timestamps, making it possible to use programs like *ls*(1) and *cat*(1) to look at a directory's historical state, but complicating the user interface and breaking things like shell filename globbing.

Finally, there has been much activity lately in providing more robust query capabilities on file systems. The standard UNIX file system supports only rudimentary query tools, like *ls* and *find*.

[SECH91] describes a strategy for doing attribute-based lookups on files, where attributes are not limited to name, size, creation time, and so forth. [SECH91] makes the point that managing the namespace of attributes is nearly as difficult as managing the namespace of files.

The *Semantic File System* (SFS), described in [GIFF91], implements attribute-based naming by allowing users to express queries as operations in the file system namespace. "Virtual" directory names may be constructed to refer to all of those files whose attributes match values in the directory name. The query mechanism is somewhat restrictive – the current implementation supports only conjuncts, for example – but the authors plan to extend the syntax.

SFS allows users to install *transducers*, or procedures that compute attribute values for particular

files. The results of these transducers can be indexed in Btrees for fast lookup later. An NFS daemon accepts requests from network clients and operates on SFS, providing these features to ordinary users without requiring them to add code to their systems.

The Inversion file system supports transactions for both user data and file system metadata. It permits finer-grained time travel than either Plan 9 or 3DFS. Like SFS, Inversion is extensible and supports indexing. It has a richer query language than SFS, but does not at present support access via NFS.

Overview Of The POSTGRES Database System

Inversion is able to provide so rich a set of services because it is built on top of a next-generation database system. This section gives an overview of the database system's architecture, with an emphasis on those feature used by Inversion.

The No-Overwrite Storage Manager

The POSTGRES database system [MOSH92] uses a novel no-overwrite technique for managing storage. This technique allows the user to see the entire history of the database and obviates the need for a conventional write-ahead log, speeding recovery [STON87]. When a record is updated or deleted, the original record is marked invalid, but remains in place. For updates, a new record containing the new values is added to the database. By using transaction start times and a special status file which indicates whether or not a transaction has committed, POSTGRES can present a transaction-consistent view of the database at any moment in history. This capability is referred to as *time travel.* Since only the start time and commit state of a transaction must be recorded in the status file, no special log processing is required at crash recovery time.

Periodically, obsolete records must be garbage-collected from the database, and either moved elsewhere or physically deleted. If the records are not saved elsewhere, some historical state of the database is lost. If time travel is desired, the records must be saved forever somewhere. This process is referred to as *record archiving.*

POSTGRES includes a special-purpose process, called the *vacuum cleaner*, that archives records. Obsolete records are physically removed from the table in which they originally appeared, and are moved to an archive.

Type and Function Extensibility

POSTGRES allows users to define new types for use in the database system. In addition, users may write functions in C or in POSTQUEL, the query language used by POSTGRES. These functions may be registered with the database system, and will be dynamically loaded by the data manager when they are invoked.

Inversion takes advantage of these two capabilities to provide strong typing on user files, and to support classification functions that describe files.

The Device Manager Switch

POSTGRES allows administrators to incorporate new storage devices by writing a small set of interface routines [STON93, OLSO92]. Based on the *bdevsw* switch in UNIX, the POSTGRES device manager switch registers the devices that are available to the database system. For each device, the required interface routines are listed. These routines are specific to the database system, and include, for example, code to create new tables and to commit transactions.

Version 4.0.1 of POSTGRES supports storage on non-volatile RAM, magnetic disk, and a 327GByte Sony optical disk WORM jukebox. The non-volatile RAM and Sony jukebox device managers operate on raw devices. In the current system, the magnetic disk device manager uses the underlying UNIX file system to store data, but a future release of POSTGRES will probably change this.

Accesses to data are location-transparent – the database manager finds the device storing the data and issues calls through the device manager switch to manipulate it. This allows users to store data on any device known to POSTGRES and manage it all identically. Logically, the database has no upper limit on its size.

The Design And Implementation Of Inversion

Inversion provides file system services to users by taking advantage of database services provided by POSTGRES. Strictly speaking, the Inversion file system is a small set of routines that are compiled into the POSTGRES data manager. Requests for file system data call these routines, which carry out the required database operations.

This section describes the Inversion support routines and how files are stored in the database system.

Decomposing Files into Tables

Files, generally viewed by users as byte streams, are stored in conventional file systems as a series of data blocks. The Inversion file system similarly "chunks" user data. Figure 1 shows the schema used to store file data in POSTGRES tables.

For every file, a uniquely-named table is created. File data are collected into chunks slightly smaller than 8kBytes. The size of the chunk is calculated so that a single record will fit exactly on a POSTGRES data manager page. This page size was chosen early in the design of POSTGRES, and was intended to make magnetic disk transfers fast. Although Inversion does not require magnetic disk in order to function, the inherited page size survives.

When a user writes a new data chunk to a file, a record is created consisting of the *chunk number*, or index of this chunk into the file, and the data chunk. This record is appended to the table storing the file. Multiple small sequential writes during a single transaction are coalesced to maximize the size of the chunk stored in each database record.

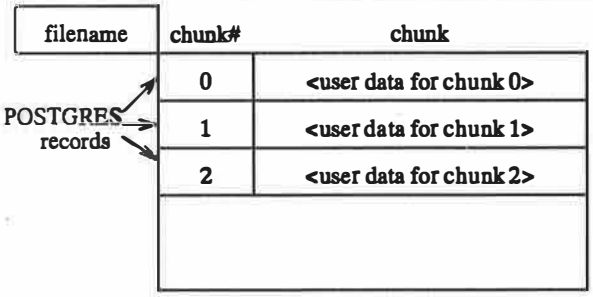


Figure 1: Decomposition of files into tables in Inversion

The Inversion file system provides a set of interface routines to create, open, close, read, write, and seek on files. Byte-oriented operations are turned into operations on chunks by calculating the chunk numbers of the affected chunks.

A file is located on particular device manager at creation. From that point on, accesses are device-transparent, both to the user and to the Inversion file system itself. The underlying device manager is called to instantiate blocks of the table storing the file.

When a file is modified, the records storing changed chunks are replaced in the normal way: the old record is marked as deleted by the current transaction, and the new record is marked as inserted by the current transaction. In order to speed up seeks on files, Inversion maintains a Btree index on the chunk number attribute.

Namespace and Metadata Management

Inversion stores the file system namespace in a table

```
naming(filename = char[],
    parentid = object_id,
    file = object_id)
```

where `filename` is the character string name of the file, `file` is the file's unique object identifier in the database (akin to an inode number in a conventional UNIX file system), and `parentid` is the object identifier of the directory containing the file.

A hierarchical namespace is imposed by having individual files point at their parent's `naming` entries. For example, the entries to construct the pathname "/etc/passwd" might be as shown in Table 1.

filename	*parentid*	*file*
/	0	810
etc	810	1076
passwd	1076	23114

Table 1: `naming` table entries for "/etc/passwd"

The root directory, named "/", appears in every POSTGRES database as shipped from Berkeley. A single database corresponds to a mount point in conventional file system architectures; all of the files stored by Inversion in a single database are rooted at "/" in that database.

Inversion includes routines to parse pathnames in order to find desired files, and to construct pathnames for particular `file` identifiers. Various Btree indices on the `naming` table speed up these operations.

Besides the file system namespace, Inversion must manage additional metadata for every file. For example, the file's owner, type, size, and last access, modification, and creation times must be recorded. These attributes are stored in the table

```
fileatt(file = object_id,
        owner = owner_id,
        type = type_id,
        size = longlong,
        ctime = time,
        mtime = time,
        atime = time)
```

where the `file` entry matches the `file` entry in the `naming` table. A simple two-way table join of `naming` and `fileatt` can construct all the metadata for a given Inversion file.

The `naming` and `fileatt` tables manage only file system metadata. The actual file data are stored in separate tables. The name of the table storing data for a particular file is computed from the `file` identifier in the `naming` table. For example, the identifier of "/etc/passwd" in Table 1 was 23114. The name of the POSTGRES table storing data chunks for /etc/passwd would be `inv23114`.

Exploiting Type and Function Extensibility

Inversion supports typing of user files. A new file type is declared by issuing a `define type` command to the database system [MOSH92]. Once this command has been issued, files may be assigned the new type. POSTGRES will automatically enforce type checking when, for example, functions are called that operate on the file.

Functions that operate on a particular type may also be registered with the database system; this applies to file types as well as to smaller database types like money and time. These functions may be invoked from the query language, and their results examined.

file type	*defined functions*
ASCII document	keywords, wordcount, linecount
troff document	keywords, wordcount, linecount, fonts, sizes
Coastal Zone Color Scanner satellite image	pixelavg, pixelcount, getpixel
Advanced Very High Resolution Radiometer satellite image	snow, pixelcount, pixelavg, getpixel, getband

Table 2: Example file types and functions

Several databases storing Inversion files exist at Berkeley storing many different types of files. For example, documentation, Hierarchical Data Format files, and images from different kinds of satellites are all stored as different file types. Table 2 lists some of the existing types and functions that operate on them. Invoking a function from the query language is easy[2]:

```
retrieve (filename)
    where "RISC" in keywords(file)
```

would find all the files stored by Inversion for which the `keywords` function was defined, and whose keywords included "RISC".

Adding new functions to POSTGRES is straightforward. Functions may be written in C or in POSTQUEL. In release 4.0.1 of the database system, these functions are dynamically loaded into the data manager process and executed with its permissions. A future version of POSTGRES will support an RPC interface to address the obvious security problems raised by this approach.

Caching and Layout Policy

Inversion does not implement cache management or layout policies independent of those used by the POSTGRES database system for regular user tables. There are two reasons for this. First, POSTGRES already implements reasonably good policies for relational data. Second, we have chosen to concentrate on providing new capabilities to clients of the file system, rather than on extensive low-level performance tuning.

This section describes cache management and layout policies implemented by POSTGRES. Inversion inherits these policies unchanged from the data manager.

[2]The example queries in this paper have been simplified somewhat for presentation. In particular, the range variables and long names used by POSTQUEL have been removed. The intent is to show how queries are expressed, not to introduce the reader to the intricacies of POSTQUEL syntax.

Cache Management

POSTGRES maintains an in-memory shared cache of recently used 8KByte data pages. The size of this cache is tunable when the file system is installed; as shipped, the system uses 64 buffers, but the version in use locally uses 300.

Data pages are kicked out of this cache in LRU order, regardless of the device from which they came. Dirty pages are written to backing store before being deleted from the cache. How they are written depends on the device backing them; for magnetic disk pages in POSTGRES 4.0.1, pages are written to the file system buffer cache, but are not necessarily forced to disk.

Individual devices are managed via the device manager switch table. Every device manager may use a private cache for its data. For example, the file system buffer cache is a secondary buffer cache for magnetic disk pages in POSTGRES.

A more interesting example of device manager caching is the Sony optical disk device manager. Due to extremely high setup costs (many seconds to load an optical platter) and relatively low transfer rates, using the jukebox directly for every transfer would be very slow. Instead, the Sony jukebox device manager caches recently-used blocks on magnetic disk. The size of this cache is tunable, and defaults to 10MBytes.

Data Layout

POSTGRES uses several strategies to improve performance by exploiting locality of reference. First, the selection of a relatively large page size (8192 bytes) means that a single data page in memory contains a good deal of user data. Second, individual device managers are free to implement sensible layout policies of their own on backing store.

Since the magnetic disk device manager uses the native UNIX file system, it inherits the layout policy used by that code. For systems that use cylinder group strategies like that described in [MCKU84], data for a single file are kept close together. As smart SCSI disks proliferate, this strategy becomes less effective, since many SCSI controllers silently remap blocks to physically distant locations on the storage medium.

The Sony jukebox device manager allocates tables in units of *extents*, where an extent is a collection of physically contiguous 8KByte data pages. The extent size is tunable when POSTGRES is installed, but defaults to 16 pages. The choice of extent size involves a tradeoff; for small tables, much of the extent will go unused, while large tables would benefit from the overhead reductions in transferring very large extents.

Services Provided by Inversion

Because it is built on top of an extensible database management system, Inversion is able to provide the following services:

- Transaction protection for changes to file data and file system metadata.
- Fast recovery after a system crash.
- Time travel on any data managed by Inversion, including metadata, data, and functions defined by users that operate on files.
- Typed files, with user-defined functions that can operate on them.
- Management of very large files.
- Strong consistency guarantees.
- Powerful query support on the file system's contents and metadata.

Transaction Protection

Transaction protection allows users to make multiple interdependent changes to a set of files atomically. For example, programmers working on a large software project may need to be able to check in several fixed source code files at the same time. If the system crashes when some, but not all, of the files have been checked in, then the software project's master directory will be in an inconsistent state.

Similarly, file system metadata changes must be made atomically. For example, when a new file is created in a directory, the directory (or file system namespace) must be updated, and the new file must be created. If only one of these operations takes place, then the file system's structure is corrupt.

Inversion supports transactions encompassing changes to arbitrary numbers of files, and commits or aborts all changes atomically. The transaction mechanism is provided by POSTGRES. No special code was required for Inversion.

In addition, a standard database two-phase locking protocol [GRAY76] allows concurrent access to files while preventing simultaneous changes from interfering with one another.

Fast Recovery

The POSTGRES transaction mechanism was designed to permit fast recovery of the database system after a crash. Data stored by Inversion is recoverable in the same way as ordinary relational data. No special boot-time file system check program needs to be run. By examining the commit state of records encountered in the database, any changes that were not committed before a system crash are automatically detected and ignored.

The only difficulties arise when the physical storage medium is damaged, or when garbage has been written to the medium by hardware or software failures. Inversion could detect these cases by making all blocks *self-identifying*; every block could be tagged with its `file` identifier and block number. Although the current version of the system does not do this, space has been reserved in the tables storing file data for this purpose.

Time Travel

POSTGRES allows users to examine any transaction-consistent historical state of the database. Since transactions are committed atomically, users can "change time" to any instant in history, and see the database exactly as they would have seen it then.

Inversion inherits this fine-grained time travel from the data manager. All old versions of files are visible. Since user-defined functions are stored in the database in the same way that ordinary files are, users can even run old versions of these functions.

Inversion does not create copies of entire files every time a change is made. Instead, only the changed blocks are saved. The appropriate historical version of a file is constructed using an index on all of the file's available data, including both old and current blocks. For files in which the user has no interest in maintaining history, POSTGRES can be instructed not to save old versions.

As was mentioned in the section on related work, both Plan 9 [QUIN91] and 3DFS [ROOM92] support time travel, but only at relatively coarse granularity. These systems snapshot file system state once a day or so. Intervening states of the file system are not visible.

The ability to see all of history can be important; for example, it allows users to undelete files removed accidentally, or to recover a working version of a program which they have changed. Inversion provides no direct support for annotating versions of files (though this capability would be easy to add as a user-defined function), but if it did, it would provide a superset of the services offered by revision control programs like *rcs*(1).

Typed Files

Conventional file systems offer little or no support for typing of files; conventions have evolved among users to deal with this. For example, C programs by convention have suffixes of ".c" on most UNIX systems.

Some systems, such as Locus [WALK83] and CODA [KIST90], support typing, but provide only a small number of file types, and do not permit users to add functions managing these types easily. [GIFF91] makes it easy to add new functions to the storage system. Inversion supports strong typing and allows users to add new functions easily. Functions operating on Inversion files may be written in C or in POSTQUEL, and will be dynamically loaded and executed on demand by the database system.

Large Files

The practical upper limit on file sizes in the current UNIX Fast File System is 4GBytes. Inversion files can be 17.6TBytes in length. Support for very

large files is important in managing scientific datasets. Researchers are currently forced to break large files up into pieces and to reassemble them inside applications; this is not necessary under Inversion.

Consistency Guarantees

Since many files have complicated structure and are semantically rich, it is important to guarantee that they remain structurally consistent. The symbol table and text space of a program, for example, contain mutually dependent entries, and neither should be changed without corresponding changes to the other. Use of transaction processing and the POSTGRES rules system [MOSH92] can guarantee this consistency.

More fundamentally, scientific users have historically managed some of their data in databases, and some in file systems. Typically, the reason for this has been that the database did not provide good support for management of large data files. This meant that records in the database could refer to files entirely outside the control of the database management code. Dividing responsibility in this way made it impossible to guarantee that references were maintained correctly. Inversion alleviates this problem by allowing users to store both tabular and file data in the same storage management system.

Query Processing

Since all Inversion data are managed by POSTGRES in tables, the user may run arbitrarily complex queries over the file system's namespace, metadata, file contents, and user-defined functions in order to find files of interest. A full-function query language makes it possible to do very sophisticated searches of the file system easily.

Services Under Investigation

In addition to the services listed above, we are exploring several other novel features in Inversion.

Users are often interested in saving only compressed versions of their files. Random access to compressed files is typically impractical; files compressed sequentially must be entirely uncompressed before random access on them is efficient.

Inversion supports compression and uncompression of "chunks" of user files. Special indices are maintained indicating the sizes of the uncompressed and compressed chunks. Random access on the uncompressed version is straightforward. Inversion determines which compressed chunk contains the bytes of interest, uncompresses it, and returns the user only the desired data. This approach provides good storage utilization and maintains reasonable random access times for files. We are investigating suitable compression strategies for the scientific data files stored in Inversion at Berkeley.

File migration is also of interest to users of very large file systems [MILL93]. Files that meet some selection criteria should be moved from fast, expensive storage like magnetic disk to slower, cheaper storage, such as magnetic tape. We are exploring strategies for using the POSTGRES predicate rules system to allow users and administrators to define migration policies. Arbitrarily complex rules controlling the locations of files or groups of files would be declared to the database manager. When a file met the announced conditions, it would be moved from one location in the storage hierarchy to another. The primary advantage of this strategy over more conventional ones is its flexibility; the rules system allows detailed migration conditions to be set up for as many different kinds of files as necessary.

Finally, distributed file systems are a subject of strong interest among both computer scientists and physical scientists. Users would like to have their files located nearby, but to have access to files stored at remote sites. For Inversion, this has implications in database cache management, migration, transaction control, and locking. Several researchers at Berkeley are exploring these issues.

Comparison to Other Database Systems

Many relational databases support *binary large objects*, or BLOBs. Typically, BLOB values can be stored in the database or fetched from it, but not manipulated from the query language in any useful way.

POSTGRES supports large object storage by creating Inversion files to store object data. All of the services available to Inversion users are also available to users of BLOBs. This includes strong typing, the ability to manipulate BLOBs from the query language, and a file-oriented interface to the data they contain. Commercial vendors of relational databases do not offer these services. Some research systems, such as Starburst [HAAS90], do offer typed large objects. Starburst provides access to large objects via an extension to the SQL cursor mechanism [LEHM89].

The integration of large database objects with Inversion means that two different clients can share data that they use in different ways. The same Inversion file can be used by a database application and by a file system client simultaneously. This means that existing programs, which store their data in a file system, continue to work. New applications can be developed that use the database directly, and can operate on the same data as the older code.

Access To Inversion Files

User files stored in Inversion may be opened, read, and written using calls modeled on those supported for ordinary UNIX files. The current implementation requires programmers to link a special library in order to access Inversion file data.

```
int
p_creat(char *path, int mode)

int
p_open(char *fname, int mode,
        int timestamp)

int
p_close(int fd)

int
p_read(int fd, char *buf,
        int len)

int
p_write(int fd, char *buf,
        int len)

int
p_lseek(int fd,
        long offset_high,
        long offset_low,
        int whence)
```

Figure 2: Interface routines for Inversion clients

The routines that manipulate Inversion files are shown in Figure 2. The important differences are in the routines `p_open` and `p_lseek`. Since the user may ask to see any historical state of the file system, the `p_open` call includes a parameter to specify the time for which the file should be viewed. Historical files may not be opened for writing. POSTGRES supports storage of objects up to 17.6TBytes in size, which means that an Inversion file may be that big. The extra parameter to `p_lseek` allows the user to specify a wider range of byte positions. Finally, the `mode` flag to `p_open` and `p_creat` encodes the device on which the file should reside at creation time.

Inversion also supports three interface new routines, `p_begin()`, `p_commit()`, and `p_abort()`. These routines begin, commit, and abort a transaction, respectively. Neither POSTGRES nor Inversion supports nested transactions, so a single application program may only have one transaction active at any time.

In the near term, we plan to provide NFS access to Inversion. In order to do so, we will be forced to support the standard interfaces for creating, opening, and seeking on files. We plan to do so, but to provide new `fnctl()` support to provide access to time travel and very large files.

However, we are unsure how to support transactions via NFS. The NFS protocol makes every operation an atomic transaction, which severely limits the utility of transactions in Inversion. We are most likely to follow the protocol specification, and to provide no multi-operation transaction protection for Inversion files accessed via NFS. Users who want the richer services may still link with the special library, and users who simply want to list directory or file contents will not need to concern themselves with transaction management.

Finally, Inversion supports ad-hoc queries on file system metadata by using the POSTQUEL query language processor. Users may run the query language monitor program to execute arbitrarily complex queries. For example, the query

```
retrieve (filename)
  where owner(file) = "mao"
     and (filetype(file) = "movie"
     or filetype(file) = "sound")
     and dir(file) = "/users/mao"
```

would return the names of all movie or sound files owned by user "mao" and found in the directory /users/mao.

Inversion currently stores several hundred satellite images from by the Thematic Mapper satellite a device which records five spectral bands for each image. A function has been written to find snow in these images. POSTGRES permits the query

```
retrieve (snow(file), filename)
  where filetype(file) = "tm"
    and snow(file)/size(file) > 0.5
    and month_of(file) = "April"
```

which will find all TM images stored anywhere in the file system which are from the month of April and which contain more than 50% snow. The `snow` function returns a count of the number of pixels that contain snow in the image. The query returns the actual number of pixels covered by snow and the name of the file storing the image.

The expressive power of a full-fledged query language is clear. However, the language can also be cumbersome and difficult for database novices to master. Although we have no plans to do so, a simpler query interface like that used by the Semantic File System [GIFF91] could be constructed. Similarly, an NFS server could manage time travel by extending the file system namespace and passing dates along to the database system for processing. This approach has been explored by [ROOM92].

Performance Of The Inversion File System

This section presents measurements of the performance of the Inversion file system. Inversion is intended to support physical scientists working on the Sequoia 2000 project [STON91], [KATZ91]. In general, these scientists use Inversion as a network file server. The system configuration evaluated here is that used by Sequoia researchers. Inversion is compared to NFS [SAND85] running on identical hardware.

System Configuration

Inversion was installed on a DECsystem 5900 with 128MBytes of main memory. The operating system running on the machine was ULTRIX 4.2.

Files were located on a 1.3GByte DEC RZ58 disk drive attached to the DECsystem 5900.

Files were opened, read, and written from a remote client running on a DECstation 3100 under ULTRIX 4.2. Client/server communication was via TCP/IP over a 10Mbit/sec Ethernet.

Inversion was compared to the ULTRIX 4.2 implementation of NFS. The NFS server was run on the same DECsystem 5900, using the same disk, as Inversion. The NFS client was the same DECstation 3100.

The NFS implementation on the DECsystem 5900 used a service called PRESTOserve to speed up writes. To guarantee that NFS servers remain stateless, NFS must force every write to stable storage synchronously [SAND85]. PRESTOserve consists of a board containing 1MByte of battery-backed RAM and driver software to cache NFS writes in non-volatile memory. As will be seen below, this substantially improved the write throughput of NFS under ULTRIX. This non-volatile memory was not used by Inversion.

The Benchmark

The benchmark consisted of the following operations:

- Create a 25MByte file.
- Measure the latency to read or write a single byte at a random location in the file.
- Read 1MByte in a single large transfer.
- Read 1MByte sequentially in page-sized units. The page size was chosen to be efficient for the file system under test.
- Read 1MByte in page-sized units distributed at random throughout the file.
- Repeat the 1MByte transfer tests, writing instead of reading.

All caches were flushed before each test. These tests measure worst-case throughput for operations that Sequoia researchers are likely to carry out.

Benchmark Results

Figure 3 shows the elapsed time to create a 25MByte file under Inversion and under ULTRIX NFS. As shown, Inversion gets about 36% of the throughput of NFS for file creation. This difference is due primarily to the extra overhead in maintaining indices in Inversion. For every page written to the file, Inversion must create a Btree index entry so that the page can be located quickly later. Btree writes are interleaved with data file writes, penalizing Inversion by forcing the disk head to move frequently. The NFS implementation does not maintain as much indexing information on the data file, and so can postpone writing its index until all data blocks have been written. This means that NFS writes the data file sequentially, improving throughput.

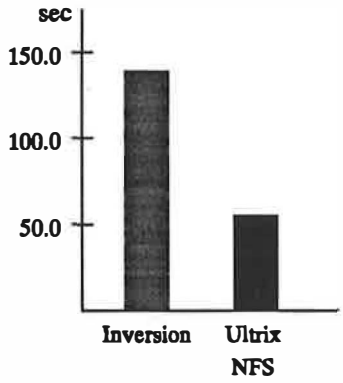


Figure 3: 25MByte file creation times

Figure 4 shows the overhead for reading or writing a single byte at a random location in the 25MByte file just created.

Since all caches were flushed prior to running the test, a disk access is required. For single-byte reads, Inversion gets 70 percent of the throughput of NFS. Single-byte writes are slightly worse; Inversion is 61 percent of NFS. Since Inversion never overwrites data in place, a new entry must be written to the Btree block index, accounting for the difference.

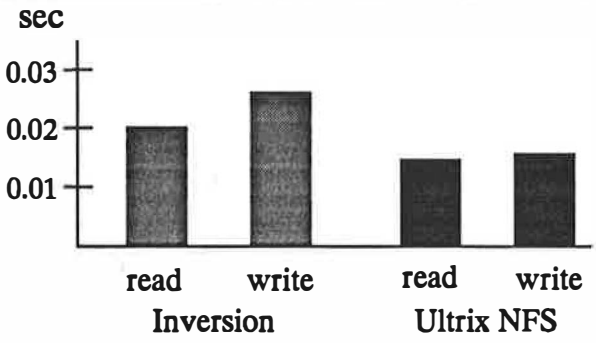


Figure 4: Random byte access

Figure 5 compares Inversion to ULTRIX NFS on large and small reads. The first pair of bars compares throughput using a single large transfer to move data from the server to the client. In this case, Inversion gets eighty percent of the throughput of NFS. When smaller transfers are used, Inversion drops to 47 percent of NFS. Profiling reveals that extra work is done in allocating and copying buffers in Inversion. If some of this overhead were eliminated, Inversion's performance could be brought closer to that of NFS. Since single-byte transfer times are much closer under the two file systems, there is reason to believe that tuning will improve Inversion.

The final pair of bars in Figure 6 compares the transfer rates of Inversion and ULTRIX NFS when the pages read are distributed at random throughout the 25MByte file. In this case, Inversion gets 43 percent the throughput of NFS. The additional overhead

incurred by traversing the Btree page index in Inversion accounts for much of the slowdown.

Figure 6 presents the write performance of Inversion and ULTRIX NFS. The tests run were identical to those performed for Figure 5, except that reads became writes. In these tests, the effect of the PRESTOserve board used by NFS is dramatic.

Since NFS must flush every write to stable storage, Inversion should have much better performance than NFS without non-volatile RAM. The reason for this is that NFS is forced to treat every write as a single transaction, and commit it to disk immediately. Inversion, however, can obey the transaction constraints imposed by the client program, and commit a large number of writes simultaneously.

Figure 6 shows that Inversion is slower than ULTRIX NFS backed by PRESTOserve. For a single large write request, Inversion gets 43 percent the throughput of NFS. For page-sized sequential writes, Inversion does worse, getting only 31 percent of NFS' throughput. For random accesses, Inversion has only 28 percent the performance of NFS. In fact, the NFS measurements show no degradation due to random accesses, since the whole 1MByte write fits in the PRESTOserve cache, and is not flushed to disk.

Evaluation of Benchmark Results

The benchmark results indicate that Inversion is penalized for not using a non-volatile RAM buffer such as PRESTOserve, and by its relatively heavyweight network communication protocol, which is based on TCP/IP.

An obvious strategy would be to disable PRESTOserve and rerun the benchmark. We used production file systems to collect the measurements shown here. Both the Ultrix and Inversion file systems are served by the same DECsystem 5900, and political considerations made it impossible to reconfigure the Ultrix NFS server for this test.

Another sensible strategy would be to run the benchmark on local file systems, so that network communication costs and the benefit of PRESTOserve were eliminated. [STON93] presents the results of such a benchmark on a 12-processor Sequent Symmetry machine running the Dynix operating system. Those results show that Inversion

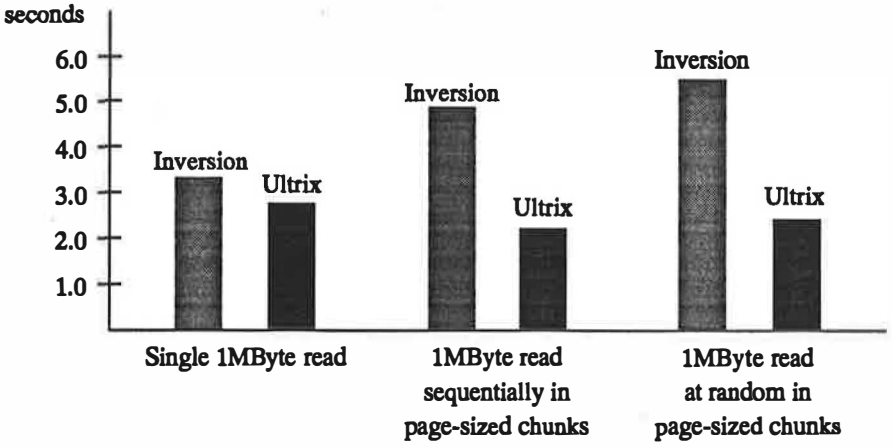


Figure 5: Read throughput

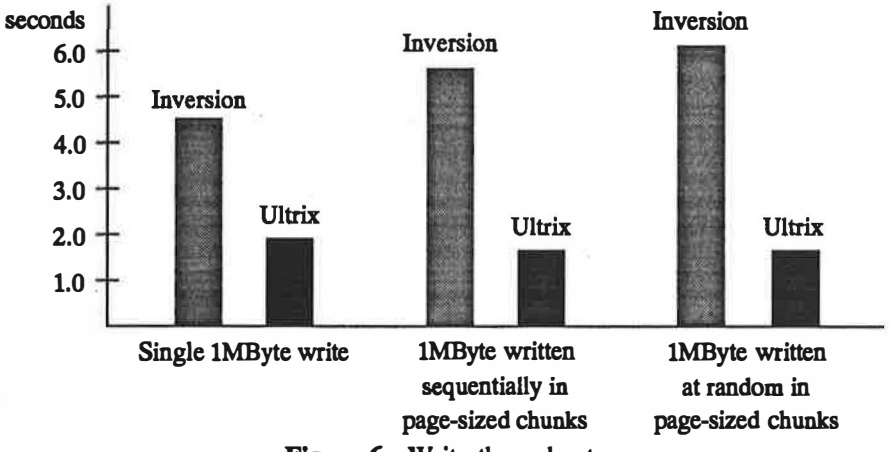


Figure 6: Write throughput

gets better than 90% of the throughput of the native file system on large sequential transfers, and roughly 70% of the throughput on small, uniformly random transfers.

A final strategy is to exploit the extensibility of Inversion to run the benchmark directly in the file system, rather than using a separate application program. The results of such an implementation are presented below.

In this case, the routines for the benchmark were declared to POSTGRES as user-defined functions, and were dynamically loaded into the POSTGRES data manager on invocation. This represents the best performance available to users under Inversion, since the benchmark and the file system are running in the same address space, and no data must be copied between them. Note that the same files can be used simultaneously by dynamically-loaded code and by the more conventional client/server architecture.

Table 3 shows the performance of the single-process benchmark. Comparable performance numbers for ULTRIX are not included, since the native ULTRIX file system does not support this approach. For convenience, the performance of client/server Inversion and ULTRIX NFS, presented in the previous section, are included.
The elapsed time for each test is reported in seconds. The measurements shown are the means of ten runs. In all cases, the standard deviation was negligible.

As Table 3 shows, the single-process Inversion benchmark is faster than either of the network benchmarks in virtually all categories. The important exception is in random write time, for which ULTRIX NFS using PRESTOserve is fastest, since no disk seeks are required. Note, however, that the single-process implementation of Inversion is faster than ULTRIX with PRESTOserve for sequential transfers. File creation is slower in both Inversion benchmarks, due to the overhead of creating the Btree index of blocks.

The important comparison is between Inversion running on two machines and Inversion running in a single process. For 1MByte operations, remote access adds between three and five seconds to the elapsed time of each test. It is clear that the client/server communication protocol used by the file system is much too heavy-weight, and should be optimized. The current implementation uses TCP/IP for communication. Given optimization of the protocol, it is reasonable to expect performance within fifty percent of ULTRIX NFS and PRESTOserve from Inversion.

Conclusions

Inversion provides significant new services to file system users by adding a small amount of code to the POSTGRES extensible database system. These services include transaction protection for file updates, fast recovery in the face of system crashes, time travel on file data and metadata, strong typing, the ability to add new functions that operate on file types to the storage system, and support for complex queries on the file system name and and attribute spaces.

The current implementation of the system requires clients to link a special library in order to use Inversion files. In the near future, we plan to extend the system with support for NFS, although the NFS interface will probably not support transactions.

Performance of the system as a network file server is reasonable, although continued tuning is certainly necessary. Depending on the access pattern, Inversion is between 30 and 80 percent as fast as the native ULTRIX file system over NFS carrying out the same operations. ULTRIX is able to exploit a large non-volatile RAM cache that is not used by Inversion, which skews performance in favor of ULTRIX. For applications in which performance is critical, users can arrange for their code to be run by the Inversion file system directly, by creating user-

Operation	*Inversion client/server*	*ULTRIX NFS*	*Inversion single process*
Create 25MByte file	141.5	50.6	111.6
Single 1MByte read	3.4	2.8	0.4
Page-sized sequential 1MByte read	4.8	2.2	0.4
Page-sized random 1MByte read	5.5	2.4	0.8
Single 1MByte write	4.6	2.0	1.4
Page-sized sequential 1MByte write	5.6	1.7	1.4
Page-sized random 1MByte write	6.0	1.7	2.9
Read single byte	0.02	0.01	0.01
Write single byte	0.03	0.02	0.02

Table 3: Elapsed time in seconds for benchmark tests in three configurations

defined functions and registering them with the database system. In this case, performance is nearly as good as the native ULTRIX file system used locally.

The Inversion installation at Berkeley currently manages approximately seven hundred megabytes of user file data, spread across magnetic, magneto-optical, and write-once optical disks. A number of special-purpose functions that operate on satellite image files have been written and are in regular use. More are under development by Sequoia 2000 researchers.

Availability

Inversion is supported in release 4.0.1 of the POSTGRES database system. POSTGRES 4.0.1 is available for anonymous ftp from postgres.CS.Berkeley.EDU (128.32.149.1) in directory pub/, as file postgres-v4r0r1.tar.Z. If you prefer to be mailed a tape, you may send a check for US $150.00 to

POSTGRES Project
557 Evans Hall
University of California at Berkeley
Berkeley, CA 94720

Attn: Claire Mosher

Be sure to specify the kind of tape you want. We are able to write 9track tapes at 1600bpi and 6250bpi, Exabyte cartridges, TK50s, and QIC tapes.

Acknowledgments

Wei Hong, Randy Katz, Ray Larson, Margo Seltzer, Mike Stonebraker, and Mark Sullivan offered guidance during early phases of the design of Inversion. John Kohl reviewed an early draft of this paper, and made useful suggestions for its improvement. Jim Frew and the entire Sequoia 2000 community have been gracious test subjects, exercising the file system and helping to identify its problems.

References

[CABR88] Cabrera, L., and Wyllie, J., "QuickSilver Distributed File Services: An Architecture for Horizontal Growth", *Proc. 2nd IEEE Conference on Computer Workstations*, March 1988.

[CARE86] Carey, M. *et al.*, "Object and File Management in the Exodus Extensible Database System," *Proc. 1986 VLDB Conference*, Kyoto, Japan, August 1986.

[CHOU85] Chou, H., DeWitt, D., Katz, R., and Klug, A., "Design and Implementation of the Wisconsin Storage System", *Software Practice and Experience*, **15**(10), October 1985.

[CHUT92] Chutani, S., *et al.*, "The Episode File System", *Proc. USENIX Winter 1992 Technical Conference*, San Francisco, CA, January 1992.

[GIFF91] Gifford, D., Jouvelot, P., Sheldon, M., and O'Toole, J., "Semantic File Systems", *Proc. 13th ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, October 1991.

[GRAY76] Gray, J., Lorie, R., Putzolu, F., and Traiger, I., "Granularity of locks and degrees of consistency in a large shared data base", *Modeling in Data Base Management Systems*, Elsevier North Holland, New York, pp. 365-394.

[HAAS90] Haas, L. *et al.*, "Starburst Midflight: As the Dust Clears," *IEEE Transactions on Knowledge and Data Engineering*, March 1990.

[KATZ91] Katz, R., *et al.*, "Robo-line Storage: Low Latency, High Capacity Storage Systems Over Geographically Distributed Networks," Sequoia 2000 Technical Report 91/3, UC Berkeley, October 1991.

[KIST91] Kistler, J. J. and Satyanarayanan, M., "Disconnected Operation in the CODA File System", *Proc. Thirteenth ACM Symposium on Operating Systems Principles*, Pacific Grove, CA, October 1991.

[LEHM89] Lehman, T., "Long Field Support in Starburst," *Proc. 1989 VLDB Conference*, Amsterdam, Netherlands, September 1989.

[MCKU84] McKusick, M., Joy, W., Leffler, S., and Fabry, R., "A Fast File System for UNIX", *ACM Transactions on Computer Systems* **2**(3), August 1984.

[MILL93] Miller, E., Katz, R., and Strange, S., "An Analysis of File Migration in a Unix Supercomputing Environment", *Proc. Winter 1993 USENIX*, San Diego, CA, January 1993.

[MOSH92] Mosher, C., *ed.*, "The POSTGRES Reference Manual, Version 4", UCB Technical Report M92/14, Electronics Research Laboratory, University of California at Berkeley, Berkeley, CA, March 1992.

[OLSO92] Olson, M., "Extending the POSTGRES Database System to Manage Tertiary Storage", Master's thesis, University of California at Berkeley, May 1992.

[QUIN91] Quinlan, S., "A Cached WORM File System", *Software – Practice and Experience*, **21**(12), December 1991.

[ROOM92] Roome, W.D., "3DFS: A Time-Oriented File Server", *Proc. USENIX Winter 1992 Technical Conference*, San Francisco, CA, January 1992.

[ROSE91] Rosenblum, M. and Ousterhout, J., "The Design and Implementation of a Log-Structured File System", *Proc. 13th Symposium on Operating Systems Principles*, Pacific Grove, CA, October 1991.

[SECH91] Sechrest, S., "Attribute-Based Naming of Files", University of Michigan Technical Report CSE-TR-78-91, January 1991.

[SELT90] Seltzer, M., and Stonebraker, M., "Transaction Support in Read Optimized and Write

Optimized File Systems," *Proc. 16th International Conference on Very Large Data Bases*, Brisbane, Australia, August 1990.

[SELT92] Seltzer, M., and Olson, M., "LIBTP: Portable, Modular Transactions for UNIX", *Proc. USENIX Winter 1992 Technical Conference*, San Francisco, January 1992.

[SELT93] Seltzer, M., Bostic, K., McKusick, M, and Staelin, C., "An Implementation of a Log-Structured File System for UNIX", *Proc. Winter 1993 Usenix*, San Diego, CA, January 1993.

[SIMM91] Simmel, S., and Godard, I., "The Kala Basket – A Semantic Primitive Unifying Object Transactions, Access Control, Versions, and Configurations", *Proc. 1991 Conf. on Object-Oriented Programming Systems, Languages, and Applications*, 1991.

[STON87] Stonebraker, M., "The POSTGRES Storage System", *Proc. 1987 VLDB Conference*, Brighton, England, Sept. 1987.

[STON91] Stonebraker, M., and Dozier, J., "Sequoia 2000: Large Capacity Object Servers to Support Global Change Research," Sequoia 2000 Technical Report 91/1, UC Berkeley, July 1991.

[STON93] Stonebraker, M., and Olson, M., "Large Object Support in POSTGRES", *Proc. 9th Int'l Conf. on Data Engineering*, Vienna, Austria, April 1993 (to appear).

[WALK83] Walker, B., *et al.*, "The LOCUS Distributed Operating System", *Operating Systems Review* v. 17 no. 5, November 1983.

Author Information

Michael Olson is a graduate student in Computer Science at the University of California at Berkeley, where he has attracted much notoriety by wearing clothes. His research interests include managing very large data stores and a categorical ranking scheme for all the local breweries. The first of these may someday lead to a Ph.D. Reach him via US Mail at:

Mike Olson
571 Evans Hall
UC Berkeley
Berkeley, CA 94720

His electronic mail address is mao@cs.Berkeley.EDU.

Operating System Support for Portable Filesystem Extensions

Neil Webber – Epoch Systems, Inc.

ABSTRACT

No standards, de facto or otherwise, exist for the programming environment found inside UNIX kernels. Yet designers hope that the VFS architecture, especially when combined with vnode stacking, will entice third parties into supplying new filesystem services such as compression or encryption. Our experience suggests that few third parties are likely to do so because of the expense inherent in supporting non-portable kernel modules in heterogeneous network environments.

We have developed kernel extensions allowing user-space implementations of such services. The extensions build on our experiences with hierarchical storage, backup, watchdogs, and vnode stacking. Our model supports common interfaces among different kernels, thus allowing portable implementations of such services.

This paper examines the VFS portability issues that inspired this work. It then discusses our solution, its relationship to other models, its relationship to vnode stacking, our implementation experiences, and future directions.

Introduction

Epoch Systems has implemented hierarchical storage management and online backup systems on a variety of UNIX platforms. Portions of our code reside inside the kernel and make extensive use of Virtual Filesystem (VFS) interfaces. Since we support this code on so many different platforms, portability and release-to-release compatibility issues for VFS modules are extremely important to us.

UNIX kernels with a VFS architecture have been commercially available for many years. Sun Microsystems, for example, described their VFS architecture in the 1986 Summer Usenix proceedings [1]. By many measures the VFS concept has been quite successful, but from a third party point of view there are two major problems:

- Few vendors have the same VFS interface.
- Few vendors provide release-to-release source or binary compatibility for VFS modules.

We call these two problems the *VFS portability* problem and the *lock-step release* problem, respectively. Together, they make VFS modules expensive to produce, expensive to port, and expensive to maintain.

To solve the VFS portability and lock-step release problems we have developed a new kernel interface, the *File Monitor Interface* (FMI), supporting portable user-space implementations of filesystem extensions such as data compression, file activity logging, hierarchical storage, and data encryption. The FMI evolved from previous Epoch hierarchical storage implementations [2, 3], watchdogs [4], and stackable filesystems [5, 6, 7, 8]. It cleanly separates the portable, release-independent, application portions of a layered VFS service from the non-portable, release-dependent, kernel portions. By providing a consistent programming environment on all systems, the FMI supports portable code in spite of VFS variability.

Of course, someone still has to implement the FMI kernel code. When a third party (such as Epoch Systems) must produce that implementation, the problems of VFS portability and lock-step release remain. But if the kernel code is simple and useful for many applications, it can be given to operating system vendors for inclusion in their base UNIX implementations. We are working with vendors interested in doing just that. Vendors who add the necessary operating system support enable construction of VFS services as true applications rather than as operating system code.

For background, this paper first explains hierarchical storage management (HSM). It then discusses VFS portability and lock-step release issues in an Epoch HSM implementation which does *not* use the File Monitor Interface. This paper then discusses the File Monitor Interface, related work, and a prototype implementation. It concludes with ideas for future work.

Hierarchical Storage Management

Hierarchical storage management (HSM) software combines high performance disk storage with other lower performance and less expensive media technologies to create a hybrid storage subsystem with high performance, high capacity, and low cost per megabyte [9]. Epoch Systems HSM software transparently migrates data between magnetic, optical, and tape storage to create the illusion

of an all-magnetic filesystem that never becomes full [2]. In essence, HSM does for disk storage what virtual memory techniques do for main memory.

At the top of the hierarchy, files reside on a magnetic disk in the native filesystem format. As that disk becomes full, the HSM software automatically selects files that have not been accessed or modified recently and transparently migrates them to a lower level in the storage hierarchy, typically to optical or tape devices. When an application later accesses a migrated file, the file is transparently brought back to the magnetic disk filesystem.

Hierarchical storage management software should *augment* native filesystems, not replace them, because native filesystems often provide special features considered important by users. Layering HSM on top of the native filesystem allows users to have HSM services *and* their native filesystem. In contrast, providing HSM only in a separate, self-contained filesystem forces users to choose between having HSM services or having their native filesystem features.

This add-on philosophy applies to other filesystem services. Consider file-by-file compression: the value of a compression module increases if it works on top of a native filesystem. The value decreases if it only operates in a separate, self-contained filesystem because it then forces users to move their files manually between the native filesystem and the compression filesystem.

Wrappers

To explain VFS portability and lock-step release issues, this section examines an Epoch HSM implementation which integrates directly into a VFS system without benefit of the File Monitor Interface.

Our current software consists of kernel-resident *filesystem wrappers*, miscellaneous kernel *glue*, and associated support daemons. Many new features have been added since the first wrapper version, but the basic architecture [3] remains intact. Figure 1 shows the wrapper code sitting between a generic VFS layer and a base UFS filesystem.

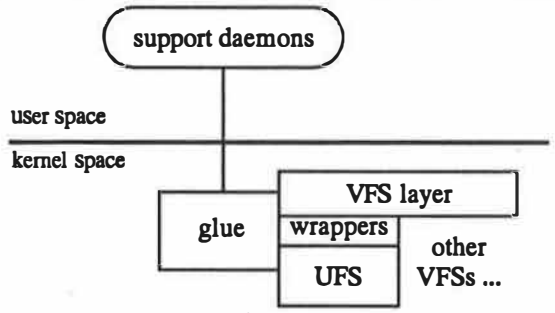


Figure 1: Filesystem wrappers

The wrapper code intercepts vnode operations and passes them either to the base filesystem or to the support daemons, based on whether the target file has been migrated. Communication with the daemons occurs via a custom pseudo-device and other glue code that Epoch adds to the kernel.

The wrappers pass some vnode operations, such as VOP_GETATTR, directly to the base filesystem. For other vnode operations, such as VOP_RDWR, the wrappers must check whether the file in question has been migrated to a lower level in the storage hierarchy. Migrated files still exist in the base filesystem but consume no data storage.[1]

An Example Wrapper

Figure 2 shows pseudo-code for a typical wrapper, VOP_RDWR, with locking and error checking omitted to simplify the discussion.

```
wrap_rdwr(vp, rw, <args>)
  struct vnode *vp;
  enum uio_rw rw;     /* RD vs. WR */
{
  int err;

  /* Outer loop for ENOSPC errors */
  do {

    if (vp is migrated) {
      /* tell daemon to get file */
      send_migfault_message(vp);
      wait for migfault reply

      /* free bitfile */
      if (rw == UIO_WRITE)
        send_freemig_message(vp);
    }

/* Let base fs do the real work */
    err = xxx_rdwr(vp, rw, <args>);

    /* Handle out-of-space. */
    if (err == ENOSPC) {
      send_outofspace_message(vp);
      wait for space
    }

  } while (err == ENOSPC);

  return err;
}
```

Figure 2: VOP_RDWR wrapper code

The wrapper has four responsibilities:

- Hide ENOSPC errors from applications.
- Detect and handle accesses to migrated files.
- Free bitfiles (explained below) during writes.
- Perform the actual VOP_RDWR operation.

[1] In a UFS filesystem, a migrated file is represented as an inode with the correct logical file size (i_size), but with no data blocks. To a kernel without wrappers it would appear to be a file with a hole covering the entire range of bytes from beginning to end.

Hiding ENOSPC

The Epoch HSM implementation prevents application programs from encountering "filesystem full" errors. To accomplish that, when the wrapper receives an ENOSPC error from the base filesystem it sends an *out-of-space* message to the support daemons and suspends the (errant) process. The daemons will then migrate old files to lower levels of the storage hierarchy, thus making space available in the filesystem. The wrapper retries the operation after the daemons have made space available.

Accessing Migrated Files

The wrapper consults an on-disk data structure (with caching) to discern when an access involves a migrated file. For accesses to migrated files, the wrapper sends a *migration-fault* message to the support daemons and waits synchronously for a reply. This has the effect of suspending the process while the daemons retrieve the file.

Freeing Bitfiles

The term *bitfile* refers to the image of a migrated file in a lower level of the storage hierarchy (e.g., on tape) [9]. Bitfiles are immutable, so operations that write to files must first restore the file to the top of the hierarchy (via a *migration-fault* as described above) and then free the corresponding bitfile. To free the bitfile, the wrapper sends a *free-migrated-image* message to the support daemons after file retrieval has completed. Unlike migration faults, the wrapper does not wait for a reply; the message is asynchronous.

Performing VOP_RDWR

The wrapper calls the base filesystem to perform the actual VOP_RDWR work. In effect, the wrapper code uses vnode stacking techniques similar to those described in "Evolving the Vnode Interface" [5] and related work [6, 7, 8].

Complications

The real wrapper code contains many complications that increase the expense of producing the wrappers and decrease their portability.

One complication involves locking. The wrappers use locks ("wrapper-node" locks) on their own data structures, like any other VFS. When executing VOP_RDWR a process will obtain, in order, both a wrapper-node lock and a corresponding "inode" (or equivalent) lock. Unfortunately, other (non-wrapper) parts of the kernel can cause the reverse locking order to occur: obtaining an inode lock before the wrapper-node lock.[2] Such paths can cause deadlocks, so the wrappers contain deadlock avoidance code and must understand the base filesystem locks (e.g., inode locks). That code adds complexity to the wrappers and reduces their portability; they can no longer stack on top of an arbitrary VFS.

[2]VFS_SYNC is one example.

A second complication involves read-ahead. The production wrappers can migrate pieces of files, not just entire files, so `wrap_rdwr` must determine whether a particular I/O will access a migrated *piece* of a file. However, it cannot determine that without knowing implementation details peculiar to the base filesystem, because the base filesystem may perform arbitrary read-ahead. Therefore, the wrappers contain code to predict the read-ahead actions of the base filesystem. Again, that code adds complexity and reduces portability.

VFS Portability Issues – Syntax

In addition to locking and read-ahead complications, there is another portability issue: VFS implementations in different kernels have little in common.

No two systems have the same vnode operation names and argument sequences. SunOS 4.1 has VOP_RDWR for both reading and writing. SVR4 has separate VOP_READ and VOP_WRITE operations. OSF/1 has VOP_READ and VOP_WRITE, but packages the error return value as one of the arguments to the macro instead of as a function return value. Some systems use VNOP_xxx instead of VOP_xxx. Last, but not least, some systems spell "vnode" differently from others.

Syntactic differences go well beyond issues of vnode operation names. Other examples include differences in kmem_alloc interfaces, differences in methods used to return errors from vnode operations, different locations for crucial data such as user IDs and group IDs, etc.

Techniques for handling such differences exist. See, for example, "#ifdef Considered Harmful" [10], which advocates creating a thin abstraction veneer to insulate code from such differences. Creating abstractions works well for application code but less well for VFS modules, because every interface between a VFS module and the rest of the kernel requires protection by an abstraction. Therefore, the thin veneer may be thin, but it is very wide. Also, not only do the interfaces change from one platform to another, but *they often change from one operating system release to another.* Keeping pace is difficult.

VFS Portability Issues – Architecture

Even after managing the syntactic issues, a third party VFS module developer faces deeper problems besides the previously discussed read-ahead and lock ordering issues. Does the kernel use a buffer cache, or a unified VFS/VM page cache? Is the kernel preemptive? Do vnodes contain locks? Is there a multiprocessor locking strategy? If so, does it serialize operations at a file level of granularity, or does it allow multiple readers? Is there a separate lock to maintain per-process read/write atomicity? In our experience, VFS modules must understand many such issues.

The problem is not that any particular VFS implementation is especially complicated or difficult to understand, but that too many VFS implementations exist. And, as with syntax, VFS architecture varies not only from one platform to another, but also from one release of the operating system to another. SunOS provides a good example: it has changed from a buffer cache architecture to an integrated page cache (SunOS 3 to SunOS 4), and later changed from a simple locking model to a fine-grained MP locking model (SunOS 4 to Solaris 2).

VFS Portability Conclusion

Developing, porting, and maintaining a VFS module is expensive. Although Epoch has succeeded with the wrapper model, the expense motivated us to search for an API that vendors would support in their systems, allowing third parties to construct *portable* add-on filesystem services.

Related Work

This section looks at related work and other solutions we chose not to pursue.

Filesystem hooks specifically for HSM

Other HSM implementations, such as the BRL/USNA Migration Project (BUMP) [11], inject specialized hooks into native filesystem code. BUMP modifies the filesystem code to trap certain operations, using one reserved IFMT encoding to indicate a *migrated file*. The modified filesystem code looks for reads, writes, truncations, etc., of migrated files and communicates them to a daemon process for handling. Other similar systems exist [12].

To use this solution to solve the VFS portability problem, one would have to ask operating system vendors to implement all the necessary kernel-resident hooks. But the kernel hooks defined by BUMP (or others) are designed for a specific flavor of HSM. They do not offer a general purpose framework for other HSM features, nor do they enable development of other interesting filesystem services. Operating system vendors understandably resist putting such specialized features into their systems.

User-space NFS servers

Since the NFS protocol provides a standard interface across multiple platforms, filesystem services can be implemented *portably* using user-space NFS servers. Also, since most vendors provide binary user-space compatibility from one OS release to another, such an implementation eliminates lockstep release concerns. The solution is also attractive because it requires no explicit cooperation from operating system vendors.

However, three issues suggested looking beyond this approach:

- Extra context switches between the requesting process and the NFS server process impose a performance penalty on local accesses.
- Extra data copy operations may be required. They too impose a performance penalty.
- All applications, even local ones, perceive NFS semantics rather than native semantics.

Nevertheless, user-space NFS servers can provide many interesting services (e.g., automounters).

There are two approaches for building user-space NFS servers. In a *filesystem* approach the server implements a complete filesystem on top of a raw disk partition or perhaps a single large file. New filesystem implementations can be debugged this way. In a *pass-through* approach the server simply translates NFS requests into corresponding system calls on files stored in a native filesystem. We generally prefer the pass-through approach because it preserves more of the base filesystem characteristics (recall the "add-on" philosophy). Note, however, that even a pass-through server still imposes NFS semantics on all its clients (including local clients).

Vnode stacking

Vnode stacking [5] layers one VFS implementation on top of another in a formalized way, with support provided by the operating system. This support replaces the ad hoc methods Epoch Systems and others use in developing their own filesystem wrappers or stacks. A lot of recent work has been done on vnode stacking, including the FICUS system at UCLA [6, 7], and the UNIX International Stackable Filesystem Working Group [8].

Vnode stacking is a powerful concept and has been used successfully to provide many services, but is not a panacea. Constructing some services, including hierarchical storage management, with vnode stacking may require non-portable architectural code and may require knowing details of the base filesystem implementation. As an example not specific to hierarchical storage management, consider a proposal for a stackable BSD quota module [5], and consider a write into the middle of a UFS file. Assume the user has reached the hard quota limit and therefore should not be permitted to allocate any more disk blocks. How does the quota module know whether to allow this write?

If there is a hole in the file at the location being written, the quota module must *disallow* the write. However, if there is not a hole at that location, the quota module should *allow* the write. The quota module has no filesystem-independent method for distinguishing the two cases in advance. The module could detect the additional disk blocks after allowing the write, but useful semantics demand detecting them before allowing the write.

OSF/1 and SVR4

A few years ago, it seemed possible that the industry might converge on some small number of kernel technologies, most notably OSF/1 and SVR4. Had that occurred, a third party vendor would have been able to gain access to a substantial portion of the UNIX market by performing just one or two ports. Although this may be true some day for application level code, it will not be the case for kernel code in the foreseeable future, because few *pure* OSF/1 or SVR4 kernels exist. Most vendors make substantial modifications to the systems they receive from OSF or USL; these systems offer no solace for those suffering from VFS portability headaches.

Our Solution

To address VFS portability issues, we developed the File Monitor Interface model supporting layered filesystem services in user-space with insulation from volatile kernel environments.

General Model

Our solution allows code outside the operating system to affect filesystem requests. The model builds on the *watchdog* concept described in [4], with additions based on our experiences with wrappers and hierarchical storage systems. Figure 3 illustrates the primary components, which include a programmable *event detection mechanism* inside the operating system, one or more *file monitors* (FMs) outside the operating system, and communication channels for *events* and *responses* between the event detector and the file monitors.

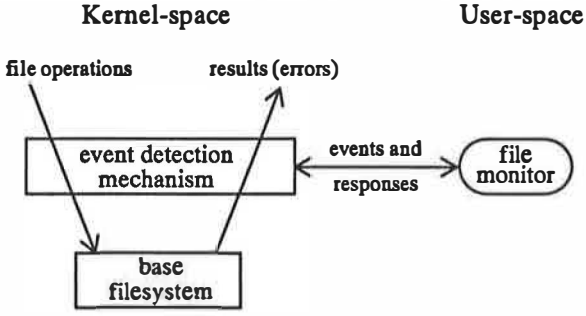


Figure 3: Event Detector and FM

The event detection mechanism traps certain file operations before they get to the base filesystem and sends them (as *events*) to a user-space file monitor. The file monitor then takes whatever actions it deems necessary and eventually returns a *response* to the kernel indicating what should happen next. In the simplest case the response indicates that the original filesystem request should continue undisturbed through the base filesystem. The event detection mechanism can also trap errors that occur after the base filesystem has processed an operation. Such "error" events function the same as the other events.

Although Figure 3 suggests a layered relationship between the event detector and the base filesystem, the interfaces visible to the user-space file monitor do not require such an implementation. Indeed, our prototype (described later) embeds event detection into the base filesystem itself.

The sections that follow describe interesting features of the event detection mechanism.

File Events

The event detection mechanism monitors files and communicates events to the FMs. File events consist of basic filesystem abstractions, such as read, write, create, etc., as shown in Figure 4. Note that these events are not specific to HSM applications. A file monitor controls which events it wishes to receive, and which files should be monitored by the event detector, via interfaces that will be described later.

```
enum fm_event {
  FE_NOP,                /* for testing */
  FE_FILE_DATA_READ,     /* read */
  FE_FILE_DATA_WRITE,    /* write */
  FE_FILE_ATTR_READ,     /* stat */
  FE_FILE_ATTR_WRITE,    /* chmod, etc. */
  FE_ERROR_RESULT,       /* ENOSPC, etc */
  FE_NAME_CREATION,      /* direnter */
  FE_NAME_DESTRUCTION,   /* dirremove */
  FE_FILE_CREATION,      /* inode alloc */
  FE_FILE_DESTRUCTION,   /* inode free */
   /* etc. */
};
```

Figure 4: Events

The semantics of events are based on vnode stacking, as suggested previously by Figure 3; data and attribute events are detected on the way down to the base filesystem, and error events are detected on the way back up from the base filesystem.

Event Responses

There are five categories of event responses: *pass-through*, *deny*, *suspend*, *implement*, and *retry*. A file monitor sends one of these responses back to the kernel after receiving an event message from the event detector.

A *pass-through* response allows the original request to pass through to the base filesystem. In other words, the filesystem request continues undisturbed.

A *deny* response returns an error code back to the originator of the filesystem operation, causing the request to fail.

A *suspend* response blocks the original request indefinitely. The suspend response may seem redundant because the request automatically becomes blocked while waiting for a reply from the file monitor. However, we have found an explicit *suspend* response important for implementing a production-quality system; this will be explained below.

An *implement* response allows the FM to perform the request implied by the event.

A *retry* response causes the operation to be started again from a point prior to the original interaction with the FM.

Handling Delays: Suspend

We introduced the *suspend* response based on our experiences with HSM implementations when a request for a migrated file could take an indefinite amount of time to satisfy. A file's inode (or equivalent) data structure must not remain locked during this indefinite delay or else something we call a *cascading inode lock* problem will occur.

One example of the cascading inode lock problem occurs during pathname lookup. This problem can be understood in terms of the *namei* algorithm shown in *The Design of the UNIX Operating System* [13]. The same principles apply in modern VFS implementations although the details vary. Consider the case of a "victim" process looking up an existing file, *abc*, in its current directory. Assume that some other process, the "perpetrator," has the *abc* inode locked indefinitely (e.g., due to migration).

The victim process begins by obtaining the inode pointer for the current directory and locking it. The victim scans the directory looking for the *abc* entry. After finding it, *and while still holding the directory lock*, the victim calls *iget* to get the *abc* inode. Iget always returns a locked inode. However, since the perpetrator already has that inode locked, the victim must sleep. Notice that the victim goes to sleep while still holding a lock on the directory containing file *abc*. This is the essence of the cascading inode lock problem: the victim now becomes a perpetrator, because it too holds an inode lock indefinitely (the directory lock) while sleeping for the *abc* inode lock. The locks gradually cascade up the directory hierarchy, and the system soon grinds to a halt because crucial directories such as */home* become part of the cascading inode lock.

An explicit *suspend* response warns the event detector about indefinite delays so it can avoid cascading inode locks. In our current prototype *suspend* partially unwinds the filesystem request and suspends it without holding inode locks.

Communication Channel

A file monitor receives event notification from the operating system via a custom communication channel in much the same way that the wrapper implementation communicates with its associated support daemons. A thin veneer defines a simple message passing abstraction containing routines such as *fm_open* to open a channel, *fm_await_packet* to wait for a message, *fm_getevent* to obtain a message from the channel, and *fm_sendresponse* to respond to a message. These routines map to *open*, *select*, *read*, and *write* calls on a simple pseudo-device in the obvious way. Hiding the pseudo-device beneath a simple veneer allows implementors to choose other communication implementations.

Programming the Event Detector

A file monitor uses new programming calls to control the event detection mechanism. The simplest call, *fm_attach*, takes as arguments a file name, a file monitor identifier, an event type (such as FE_FILE_DATA_READ), and a communication policy (synchronous or asynchronous). The call establishes a persistent relationship between the file and the specified file monitor.

To be useful for HSM services, as well as many other services, the relationship set up by *fm_attach* must be stored persistently by the filesystem.[3] To make the event detection mechanism useful without requiring persistent storage for file monitor specifications, two other event detector programming calls exist: *fm_glbl_attach* (global attach) and *fm_dflt_attach* (default attach). Both calls take the same arguments as *fm_attach* but do not require persistent storage. The *global attach* function attaches the described monitor to every active (in-memory) vnode object in a particular filesystem. The *default attach* function specifies the event detection profile to use as new vnode objects become active (in-memory) in a particular filesystem. Other miscellaneous calls, such as *fm_stat*, allow applications to obtain the current event detection status of files.

An Epoch HSM service can use these functions to set up monitors for file reads, file writes, file errors (for ENOSPC), and several other events. The first time an I/O operation occurs on a newly activated file, the file monitor receives an event notification. If the file is fully resident on the magnetic filesystem, the file monitor knows that no further intervention for reads and writes will be required and can turn off the file read and file write events, while leaving the file error event in place (for ENOSPC detection).

Having both a persistent mechanism (fm_attach) and a non-persistent mechanism (fm_glbl_attach and fm_dflt_attach) allows operating system vendors to make their own choice regarding implementation cost versus performance. High performance requires that the kernel itself store some persistent state regarding the event detection profiles of individual files; that way the kernel can often bypass file monitor communication. On the other hand, a vendor can choose to only support the non-persistent control mechanisms, which leads to lower performance but places less burden on the base filesystem. The performance data we have gathered from our prototype implementation suggests that the cost of contacting the file monitor once when a vnode becomes active may not be prohibitive. It becomes even less of a problem if the vnode cache

[3] Extended file attributes are useful for this.

policies allow that cost to be amortized over a number of file accesses.

Name Space Events

Because we are interested in developing an architecture useful for a variety of services, we include other filesystem events beyond those needed for HSM. Name space events, which allow programs to learn about renames, links, and unlinks, are one example. Among other things, they can be used to construct reliable online backup programs.

Directory renames cause problems for online backup programs. The worst case occurs during an active backup run when a directory moves from a region of the filesystem the backup program has not yet descended to a region that the backup program *has* already descended. In that case, the backup misses all the files (and subdirectories) underneath the renamed directory. Subsequent incremental backups will not correct the situation because timestamps on the affected (missed) files will generally contribute to the appearance that the "problem" backup covered them and so the incremental backup will not include them.

Name space events provide mechanisms for dealing with such problems. The event detector can generate an event whenever directory entries are added, deleted, or changed. A file monitor can use the events to implement one of two different schemes. In a *locking* scheme, the file monitor suspends the process making the change, thus keeping the filesystem stable during backup scanning. This is similar to the approach used by Sun's CoPilot backup product [14]. In a *non-locking* scheme, the file monitor communicates the event to the backup program, which can then take appropriate action. This appears to be a more general solution than the locking scheme; however, we have not yet tried implementing a backup program that uses the non-locking scheme.

Event Detection as an OS Feature

The new file monitor architecture generalizes the old wrapper architecture. Special purpose mechanisms that were only useful for one application (Epoch style HSM) have been replaced by general mechanisms useful for many applications. Policy decisions that had been compiled into the wrappers, such as the decision that deleting a bitfile can proceed asynchronously with respect to file writes, have been replaced with mechanisms for controlling such decisions at run time.

Once generalized, the kernel code becomes attractive for direct incorporation into operating systems. Two benefits accrue after a vendor takes over ownership of the kernel code. First, problems of lock-step release and VFS portability become less serious because the vendors own all the kernel code and are in a much better position to maintain it as their kernels evolve over time. Second, the mechanisms become available for use by other third parties. By putting the mechanisms into their kernels, vendors enable development of add-on filesystem services such as HSM, file compression, encryption, etc.

New System Calls

We have also defined a small set of new system calls for general application use. The calls extend UNIX in ways that ease the construction of meaningful file monitors. However, many of the calls are useful extensions in their own right even without an event detection mechanism. The discussion below describes these calls.

Invisible Read/Invisible Write

Hierarchical storage systems use file timestamps to choose migration candidates. Unfortunately, if the HSM software uses the standard *read*() system call when copying files off the magnetic disk, then the very act of migrating a file will update its access time, making the file a good candidate for bringing *back* to the top of the hierarchy.

More generally, any backup program that reads files through the filesystem interface will disturb file access times. Even without HSM software this can be a concern. To avoid disturbing access times, some filesystem utility programs have gone to the extreme of bypassing the filesystem entirely; they can read files off the raw magnetic disk.

To solve the access timestamp problem we defined two new system calls, *invisibleread()* and *invisiblewrite()*, to manipulate files without updating timestamps. Only processes with appropriate privileges (usually *root*) may use them.

We have also experimented with using a new *open*() flag to indicate invisible access. The most general solution (one we have *not* tried) consists of an alternate set of timestamps and an *open*() flag indicating that I/O should update the alternate timestamps instead of the standard ones. That allows backup and migration programs to manipulate files without disturbing the ordinary timestamps, and also addresses security and accountability concerns raised by the invisible-read and invisible-write interfaces. All operating systems should have some method of distinguishing backup-oriented file access from ordinary file access.

File Handles

Some file monitors need access to the monitored file during event processing. For example, a *stat*() call on the monitored file allows the file monitor to determine the owner of the file. Without that capability, the file monitor would have to maintain its own parallel copy of file ownership data.

Instead of relying on pathnames to support such access (the *watchdogs* system solution [4]), we use an *open-by-handle* interface. Handles, similar to

those used by NFS server implementations, are exported to user-space processes. A new system call, *open-by-handle*, allows suitably privileged processes to convert such handles into open file descriptors. This gives file monitors access to files without requiring that the kernel track or compute full pathnames.

The event packets generated by the event detection mechanism include a file handle that the file monitor uses to access the file via the open-by-handle interface. (The kernel filters out events generated by the file monitor itself, to prevent infinite recursion.) More generally, any process can make use of file handles for a variety of purposes. Two additional interfaces, *fd-to-handle* and *path-to-handle*, produce file handles from file descriptors and pathnames, respectively. Even without an event detection mechanism file handles are useful, for example, when implementing a *pass-through* NFS server in user-space.

Implementation

This section describes a prototype implementation of the event detection mechanism. All prototype work has been done with SunOS 4.1.2 running on a Sparcstation IPC.

Per-vnode event detection information is stored in an in-memory structure called an *event_detector* structure, shown in Figure 5. A new field was added to the generic vnode to hold a pointer to this structure; a NULL pointer indicates no monitoring. The implementation allows multiple vnodes to share references to a single copy of the event_detector structure for vnodes that have identical event detection profiles.

```
struct event_detector {
    unsigned long eventmask;
    unsigned long syncmask;
    int           holdcount;
    struct event_tuple
          eventarray[ NEVT ];
};
```

Figure 5: Event Detector structure

The structure contains a holdcount to record the number of vnodes pointing to it. Two bit masks, eventmask and syncmask, summarize the events being monitored. The eventmask contains one bit for each type of event. The syncmask contains one bit for events using synchronous communication. A simple array, eventarray, holds the tuples describing the monitored events, with one tuple per event type to record the monitor ID and the communication policy. The prototype only permits attaching one file monitor to a given vnode for any particular event, although it allows attaching different monitors to a given vnode for *different* events. In other words, one file monitor per event type, per vnode.

To determine whether or not a given vnode requires intervention on a particular event, a macro, VP_MON, implements the following checks: If the event_detector pointer in the vnode is NULL no intervention is required. Otherwise, the eventmask in the event_detector structure is checked for a bit corresponding to the event in question.

Generating Events

We implemented the event detection mechanism by modifying UFS to generate the necessary file events at key points. In SunOS 4.1.2, the key point for file read operations is inside the ufs_getapage() routine. The key point for file write operations is inside the ufs_putpage() routine. We added simple calls in both routines to generate the appropriate file monitor events. The calls package up the inode handle, file offset, and I/O length information and send them to a file monitor. Figure 6 shows an example code fragment.

Finding the correct location in a given filesystem to inject this call is crucial. The location must be close enough to the actual I/O path to be immune from confusion due to read-ahead policies and such. Yet it must still be possible to recover the file-relative offset and length information; at locations too low in the implementation layering that information is often unavailable.

```
/*
 * This is the code injected into
 * ufs_getapage, at the point where
 * it is about to set up an I/O for
 * a given range of bytes in a file.
 */
if (VP_MON(ITOV(ip), FE_FILE_DATA_READ))
  rv = fm_send_rdwr(ITOV(ip),
                    FE_FILE_DATA_READ,
                    offs, len);
```

Figure 6: Sending a read event

Multiple File Monitors

The prototype allows different FMs to operate on different events for a single vnode, but does not permit two different FMs to operate on the *same* event for a single vnode. This restriction is due to the event_detector data structure, which only stores one event tuple per event type, per vnode. A more general implementation could store a list of event tuples per event type, per vnode.

In a stackable vnode implementation a second method becomes possible: each stacked event detection mechanism can have its own simple one-monitor-per-event-type data structure. Multiple monitors would then be implemented by pushing multiple instantiations of the event detection mechanism onto the vnode.

The stacking approach has the advantage of factoring out the complexity of multiple monitors by

exploiting the power of vnode stacking. The drawback is that an event detector implemented in a pure stacked method resurrects all the layered VFS issues (locking, read-ahead, subtle dependencies on the base filesystem, etc.).

Miscellaneous Implementation Problems

Two other implementation issues deserve mention: hostile ENOSPC behavior, and NFS interactions.

Hostile ENOSPC Behavior

Many filesystems have hostile ENOSPC behavior. For example, the BSD4.3 UFS code prints a "write failed: file system full" message on the system console, in the system error log, and on the user's terminal when the disk becomes full. That behavior prevents a layered filesystem service from intercepting the ENOSPC error return and transparently repairing the problem. For example, a simple ENOSPC retry service could suspend the affected process for a minute or so and then retry the operation in the hope that the condition fixes itself. A more useful ENOSPC service could take active steps to make space available on the target filesystem; indeed, Epoch HSM software does just that. Unfortunately, the ENOSPC service has no way to erase the disconcerting "write failed" error message from the user's terminal. Layered filesystem services raise many such issues.

NFS Interactions

The second implementation problem comes from unfortunate interactions between NFS implementations and HSM systems. In many scenarios, an NFS operation on a migrated file consumes an *nfsd* for an unbounded period of time. For example, an NFS read may take all weekend to complete if it needs access to an optical volume not currently loaded in an automated library unit (jukebox). Even when all volumes are available, competition for optical drives can introduce long delays during migration storms. A single client can consume *all* the NFS server processes by performing a few parallel requests for migrated files. Once all *nfsd* processes are blocked awaiting access to migrated files, no other clients can access the NFS server!

We solve this problem by determining whether or not a given HSM request comes from an NFS server daemon. Migration requests that do *not* come from an NFS server daemon context are suspended according to the methods described throughout this paper. Requests that *do* come from an NFS server daemon context are simply dropped if they are not satisfied in a timely fashion. For example, the wrapper implementation never puts an nfsd to sleep awaiting migration-fault service, but instead makes the nfsd drop the request.[4]

[4]The policy is actually slightly more complex than that; the wrappers only cause the nfsd to drop the request if the request is not satisfied within a tunable period of time. Also, how one convinces an nfsd to drop the request without returning an error to the client varies from one system to another.

In the file monitor architecture the *suspend* reply that a file monitor can return to the kernel is crucial for solving these types of problems. The suspend reply alerts the kernel that the operation may take a long time. The kernel can then decide whether to suspend the affected process (the normal case) or ask it to forget about the operation (the nfsd case). Note that in the file monitor architecture, the kernel-specific code that makes this decision gets implemented once, in the event detection and response mechanism, rather than having to be duplicated throughout every add-on filesystem service.

Performance Issues

There are three primary performance issues for mechanisms such as file monitors:

- **Global cost:** The performance cost imposed by the event detection mechanism on *all* filesystem operations.
- **Partial use cost:** The performance cost imposed on accesses to a file when the mechanism is in use, but is not triggered by a particular operation. For example, the read/write performance penalty when only error events are being trapped determines the viability of an ENOSPC service.
- **Full cost:** The full performance cost incurred by operations to a monitored file that *do* contact the file monitor.

We measured these costs on a Sparcstation IPC running our SunOS 4.1.2 prototype event detector.

The *global cost* and the *partial use cost* depend on the VP_MON macro implementation. In the general case, the VP_MON macro compiles into twelve instructions. In many cases, the event type passed into VP_MON is a constant and VP_MON becomes nine instructions. In either case, the performance overhead is minor.

To measure the *full cost*, we modified *getpid*() to send synchronous FE_NOP messages and wrote a C program to call *getpid*() in a loop. Our prototype implementation can send 1600 such "speed of light" messages per second for an overhead of about 625 microseconds per event. We have not yet analyzed where all the time goes but suspect, based on prior kernel experience, that context switching consumes the bulk of it. No effort has yet been made to optimize the prototype.

Many filesystem services are insensitive to such overhead. For example, accessing a migrated file typically involves a many-second delay for robotic movement in an automated library unit. It will be difficult to even *measure* a 625 microsecond

overhead in such an application. Compression and encryption applications have similar characteristics, though in those cases the overhead is related to the computational algorithms rather than to physical device limitations.

Future Work

We are pursuing further development work jointly with our OEM partners. We are also building a reference implementation of the event detection mechanism using a commonly available UNIX source base. Future ports of Epoch Systems HSM technology and backup technology will proceed cooperatively with platform vendors; the vendors will implement the event detection mechanisms in their systems and we will provide a file monitor that implements our HSM algorithms. As we gain more experience with this model we hope to encourage other third parties (including Epoch competitors) to use these interfaces. Epoch's experience demonstrates a strong market for data management services; file monitors should help UNIX meet those needs.

We have also investigated hierarchical storage integration for other operating systems and believe the file monitor concept will be useful even outside the UNIX arena. Finally, we have used specialized extended attribute mechanisms in our HSM software but would like to do more experimentation with generalized filesystem support of extended attributes.

Acknowledgements

Greg Kenley provided the original inspiration for this work. Other contributors include Bob Burchfield, Sheila Coleman, Antony Foster, Ross Garber, Bob Israel, Laura Israel, Rob Kenna, Tracy Taylor, Noelie, Whoopi (holding things down), John Wallace (pizza), Lam To (taking over my most recent project), and Mark Hecker (taking over my second-most recent project).

References

[1] Steven R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," *Proceedings of the Usenix 1986 Summer Conference*, Atlanta, GA, Summer 1986, pp. 238-247.

[2] Tracy Taylor and Rich Fortier, "Using Optical Disks to Extend the Capacity of Magnetic Disks Through Hierarchical Storage," *Uniforum 1989 Conference Proceedings*, March 1989.

[3] Robert K. Israel, Antony W. Foster, Arun Taylor, Tracy Taylor, Neil Webber, "Evolutionary Path to Network Storage Management," *Proceedings of the USENIX 1991 Winter Conference*, Dallas, TX, January 1991, pp. 185-197.

[4] B. Bershad, C. Pinkerton, "Watchdogs: Extending the UNIX File System," *Proceedings of the USENIX 1988 Winter Conference*, Dallas, TX, February 1988, pp. 267-276.

[5] David. S. H. Rosenthal, "Evolving the Vnode Interface," *Proceedings of the USENIX 1990 Summer Conference*, June 1990, pp. 107-118.

[6] John S. Heidemann, Gerald J. Popek, "A Layered Approach to File System Development," *Dept. of Computer Science UCLA, Technical Report CSD-910007*, March 1991.

[7] John S. Heidemann, "Stackable Layers: an Architecture for File System Development," (M.S. Thesis) UCLA, *UCLA technical report CSD-910056*, March 1991.

[8] UNIX International Stackable Files Working Group, "Requirements for Stackable Files," *Unix International*, Parsippany, NJ, June 30, 1992.

[9] IEEE Technical Committee on Mass Storage Systems and Technology, "Mass Storage Systems Reference Model: Version 4," May, 1990.

[10] Henry Spencer, Geoff Collyer, "#ifdef Considered Harmful, or Portability Experience With C News," *Proceedings of the USENIX 1992 Summer Conference*, San Antonio, TX, June 1992, pp. 185-197.

[11] Michael J. Muuss, Terry Slattery, Donald F. Merritt, "BUMP: The BRL/USNA Migration Project," *Unix and Supercomputers*, 1988.

[12] P. Michael Haffley, R. Bruce Renda, et. al., "Hierarchical File Systems and Sparcus' Strategy," *Sun User Group 1991 Proceedings*, 1991.

[13] Maurice J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, Englewood Cliffs, NJ, 1986, pp. 74-76.

[14] Steve Shumway, "Issues in On-line Backup," *Proceedings of the USENIX Fifth LISA Conference*, San Diego, CA, September 1991, pp. 81-88.

Author Information

Neil Webber is a Consulting Engineer at Epoch Systems, where he has been working with UNIX kernels for the past six years. His contributions include the 4.3BSD UNIX kernel port for the Epoch–1 platform, filesystem "wrappers" for SunOS based migration software, miscellaneous kernel code for the Epoch–2 software products, participation in UI working groups and OSF SIGs, and a smattering of actual application code here and there. Prior to Epoch he worked on real-time operating systems at Automatix Inc., a machine vision and robotics company. Reach him electronically at nw@epoch.com, or via U.S. Mail at Epoch Systems Inc., 8 Technology Drive, Westborough, MA 01581.

File Systems in User Space

Paul R. Eggert – Twin Sun, Inc.
D. Stott Parker – UCLA Computer Science Dept.

ABSTRACT

Current methods for interfacing file systems to user programs suffer two major drawbacks: they require kernel modifications or root privileges, and they are too complicated to be given to ordinary users. In this paper we show alternative methods are possible. The recent rise of dynamic linking provides a new way for users to develop their own file systems: by interposing a layer of user code between a program and the system call interface, a user can alter or extend a file system's behavior.

For greatest power and reliability, such changes to file system behavior must be managed systematically. We present two user-extensible file systems that are loosely modeled on intensional logic. IFS0 is simple, and supports only extended pathname interpretation for files: it permits certain shell-like expressions as pathnames. To this, IFS1 adds directory lookup and an escape mechanism for interpreting pathnames that can be modified by the user at any point. These file systems operate by modifying the semantics of UNIX system calls that take pathname arguments.

With IFS1 a user can develop a wide range of useful file systems without writing a line of C code. We have developed a variety of sample file systems with IFS1, including `tar` image navigation and a software development file system resembling 3DFS. IFS1 can thus be thought of as a simple user-programmable file system toolkit.

Introduction

The past few years have seen the development of hundreds of file systems; everything from AFS (Andrew File System) to ZFS (Zebra File System). The high level of interest warranted a USENIX workshop [1] on the subject. By and large, these file systems have been developed to satisfy user needs not well addressed by traditional UNIX file systems.

Unfortunately, existing file system interfaces discourage user experimentation. Changing file systems requires kernel modifications or root privileges; the task of writing a new file system is far too complicated to be given to most users. Current file system interfaces discourage adding even simple naming services. For example, users often complain that applications do not understand pathnames like "`~`" or "`$HOME`" that use shell-interpretable expressions. Many applications have independently added support for these expressions, but this is a wasteful duplication of effort. It's hard to imagine solving this problem with current systems, since these abbreviations depend on information not readily available to either the kernel or the file system.

We propose intensional file systems as a simple, easy-to-explain abstraction for implementing one file system on top of another, an abstraction that provides motivating design principles for file system designers, and that makes it easy for users to define new naming rules. In an intensional file system, a file system is stored as a description of how to name or produce files, instead of as the filenames or contents themselves. The term "intensional" comes from intensional logic, which concentrates on the problem of ascribing meaning to a logical expression or name. The fundamental design problem of an intensional file system is to specify the rules and mechanism for "extensionalizing" the files of an intensional file system, i.e., for computing the actual file that an IFS file denotes.

Our main motivation for intensional file systems is to provide a simple, systematic method for users to augment their file systems with naming services. Our design principles are heavily influenced by those of Satyanarayanan [12] for modern distributed file systems. We assume that clients have CPU cycles to burn. Our goals are to cache on clients whenever possible, to let users exploit usage properties, to minimize system-wide knowledge and change for easy administration, to minimize the security risk, and to have adequate performance.

Naming and Intensionality

Many file systems amount to either an *extended name service* or an *extended system call service*. The former provide a richer view of the underlying files, but little more than this; examples include the Semantic File System [5], 3DFS [8], and TFS [7]. The latter extend many system calls beyond their normal UNIX semantics.

This work was supported by a University of California/Twin Sun MICRO grant, and NSF grant IRI-8917907.

The central issue in the first kind of system is what a "name" is, and how it is to be interpreted. The issue in the second kind of system is the mapping between a particular system call and its semantics. Philosophers have tried to clarify issues in mapping between expressions and their meaning for several hundred years. This work is usually lumped under the subject "intensionality" or "intensional logic".

Intensionality

Intensionality ascribes meaning to a sentence or a expression (such as a name). The *intension* of an expression is the procedure that finds the value (concrete meaning) of the expression in any given context. This value is also called an *extension* of the expression. Briefly, intensionality makes meaning be relative to context.

The meaning of expressions are often context-dependent: the meaning of the phrase *the current time* depends on when and where it is evaluated, and the sentence *The expression* `$(E)` *has the same meaning as the expression* `'E'`. has the meaning *true* for the Korn shell, but *false* for the traditional Bourne shell.

Intensional logics formalize these distinctions, and distinguish between expression syntax, intension, and extension. Syntax defines the expressions of a language; the *extension* of an expression is its value or what it denotes; and the *intension* of an expression is a function on contexts that yields the extension in the given context. The intension of an expression is a function from a context to a value. Figure 1 shows that an intension is the connection between a language expression and its extension.

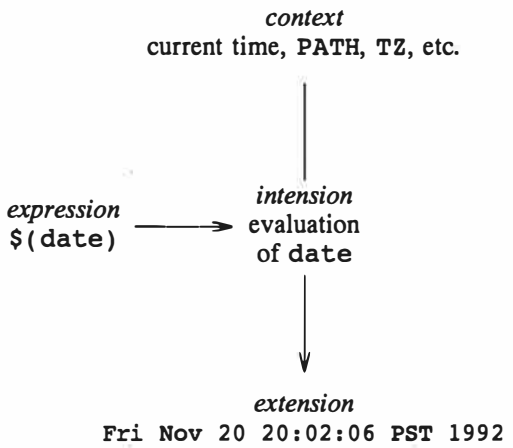


Figure 1: Intensions map contexts to extensions

For a clear, up-to-date discussion of the philosophical perspective on issues in intensionality, see Zalta [15].

Intensionality and File Systems

Naming mechanisms in file systems have been ad hoc, and could benefit from a more general foundation. A naming mechanism must somehow cover the distinctions just raised if it is to be general.

Intensional logic suggests a foundation. However it must be adapted for file system applications for a variety of reasons:

- Names in English (or any other natural language) are usually simple proper names, while names in file systems are complex paths with multiple atomic components, and can be highly time- and context-dependent.
- The concept of "evaluation" in English is limited, but is sophisticated and heavily used in an operating systems context.
- File systems raise many issues that are not of concern in processing English, such as performance, security, and conceptual simplicity.

Intensionality mechanisms used in file systems should reflect concerns like these. In IFS, we adapt the idea of intensionality in two ways:

1. The space of pathnames can be extended to include not just ordinary filenames, but also expressions. Any general name service can then be built by providing an interpreter that evaluates name expressions.
2. The same idea can be applied to system calls. Any system call (like `chmod("-",0755)`) is an expression, whose interpretation can depend on context. We provide an interpreter that defines the context, i.e., defines what the system call does. This gives a general system call interface.

The second treatment is more general than the first, since name service is essentially determined by the `open` system call.

We have built two file systems that represent these two adaptations of intensionality in a user-programmable way. IFS0 is a name service that permits shell-like expressions as filenames, while IFS1 adds directory lookup and user-defined naming escapes.

IFS0

We initially conceived IFS0 [4] as a response to the evolution of proliferation of "buildable files", taking advantage of the fact that today CPUs often have cycles to burn.

Today many files (formatted manual pages, diagnostics, binary executables, `tar` files, files copied from across the net, etc.) can be easily produced on demand whenever needed, rather than being stored on disk. Our idea was to somehow include a `make`-like capability into the file system, allowing it to rematerialize a buildable file whenever the file is needed. This idea grew into IFS0.

Overview of IFS0 Use

IFS0 extends the UNIX tree-structured pathnames by adding intensional pathnames, which are expressions evaluable with the shell. Intensional pathnames can evaluate to either file contents, or to names of other files. Pathname expressions evaluating to file contents are written with enclosing parentheses `(...)`, while expressions evaluating to pathnames are written using the Korn shell notation `$(...)`. For example, the intensional pathname `(date)` evaluates to the current date, while the intensional pathname `$(whoami)` evaluates to the file named by the output of the command `whoami`.

IFS0 is quite simple from the user's viewpoint. With IFS0 the user creates an intensional file using the ordinary UNIX `ln` command, e.g.,

```
$ ln -s '(date)' now.txt
$ cat now.txt
Sun Mar  8 20:06:31 PST 1992
$ awk '{print $4}' now.txt
20:07:01
```

To see whether a file is intensional, you invoke `ls -l` in the usual way. The ownership, length, and date is that of the intension, not the extension. To see the extension, append the `-L` option; just as with symbolic links, it causes IFS0 to refer to the extension instead of the intension.

```
$ ls -l now.txt
lrwxrwxrwx 1 pre   6 Mar 8 20:06 now.txt -> (date)
$ ls -lL now.txt
-r-------- 0 pre  29 Mar 8 20:14 now.txt
```

For a more practical example, consider the following `make` rules:

```
dist.tar.Z: dist.tar
        compress dist.tar >$@
dist.tar:
        tar cf - *.h *.c >$@
clean:
        rm -f dist.tar*
```

These can easily be replaced by intensional files that are always up-to-date, takes far less space than their extensional counterparts, and never need cleaning, as follows:

```
ln -s '(compress <dist.tar)' dist.tar.Z
ln -s '(tar cf - *.[ch])' dist.tar
```

There is no problem with defining intensional files in terms of other intensional files this way. For example, to make a compressed tape, one can execute the command

```
dd <dist.tar.Z >/dev/rst1
```

– in the process of extensionalizing `dist.tar.Z`, IFS0 recursively extensionalizes `dist.tar` first.

Intensional Pathnames in IFS0

Any tree-oriented file system provides *extensional pathnames*, sequences of existing filenames specifying a path to a desired file from some initial directory. IFS0 extends this widely-used model by permitting *intensional pathnames*, which include shell-evaluable expressions.

Intensional pathnames are essentially shell programs that yield a pathname when evaluated. For example, the intensional pathname `$(sunp Xsun X)` yields the name that is the output of the shell command `sunp Xsun X`. Suppose that the `sunp` command is defined as follows:

```
#!/bin/sh
if (sun) 2>/dev/null
then echo $1
else echo $2
fi
```

Then the pathname `$(sunp Xsun X)` is equivalent to the pathname `Xsun` if interpreted on a Sun workstation, and otherwise is equivalent to the pathname `X`.

Extensional pathnames are annoyingly restrictive, and this has led to the creation of symbolic links, conditional symbolic links, and other similar extensions to UNIX. IFS0 lifts the restriction in a general way, letting pathnames include expressions. Although symbolic links were regarded with caution when they were introduced, they are now a popular mechanism in UNIX. IFS0 can be used as a natural extension of symbolic links.

Specifics of IFS0 Pathname Interpretation

IFS0 overloads the conventional notion of *pathname* with two specializations: *extensional pathname* and *intensional pathname*. An extensional pathname is a standard UNIX pathname.

In IFS0, an *intensional pathname* is of the form

D/$(*command*)

where *D* is an extensional pathname. When this pathname is interpreted, *command* is executed in the *D* directory, and the output is interpreted as a pathname. The syntax is borrowed from the Korn shell.

Mirroring these pathname types are file types. An *extensional file* corresponds to a conventional UNIX file. An *intensional file* is a symbolic link containing an *extensional pathname* or *intensional pathname*. When opened, intensional files cause evaluations that ultimately yield an extensional file.

IFS0 redefines *pathname system calls*, i.e., system calls like `open` and `stat` that have pathname arguments and require extensional files. IFS0's interpretation of these calls differs from that of the underlying file system in the following ways:

1. In IFS0, the pathname can be intensional. If so, it is evaluated before further processing.

2. Symbolic links can contain intensional pathnames. Therefore, pathname system calls first invoke `readlink` to see whether the file is intensional.
3. The context for an `open` includes not only the working directory, the user id, and so forth, but also includes anything accessible to the process that interprets the intensional pathname.
4. Accessing a file repeatedly uses the IFS0 evaluation mechanism until it yields an extensional file, or an error.
5. Extensionalization aborts if if the total number of dereferences required exceeds the declared limit (currently 20). The limit of 20 prevents loops, and was inspired by the traditional limit of 20 symbolic link expansions in Berkeley UNIX.

IFS0 also supports one abbreviation. The pathname

D/(command)

is equivalent to the name of a temporary file containing the output of *command*.

IFS1

IFS0 showed us the potential for programmable file systems interfaces. Although IFS0 is satisfactory as a file system by itself, it has two major weaknesses.

First, IFS0 insists on having an intensional file (symbolic link) for any object file that one wants extensionalized. For example, with IFS0 we can store a large file `F` in compressed form `F.Z`, and use its extensional form:

```
$ ln -s '(zcat F)' F
$ ls -l F*
lrwxrwxrwx 1 pre      17 Nov 10 22:00 F -> (zcat F)
-rw-r--r-- 1 pre  788213 Nov 10 22:00 F.Z
$ ls -lL F
-r-------- 0 pre 1297200 Nov 10 22:01 F
```

Unfortunately, with IFS0 we must manually create the symbolic link `F` in order to get access to the uncompressed file.

Second, IFS0 extensionalizes only the last filename component of a pathname. This simplifies the implementation, but it prevents intensional directories, which are highly desirable. We were reluctant to add intensional directories to IFS0, since it would require one extra `readlink` system call per pathname component, even for ordinary extensional paths.

When we considered fixes for these shortcomings, we were struck by the wealth of options available. When extensionalizing a directory, should the resulting directory temporary be placed in `/tmp` or near the intensional directory? Should extensionalized directories be shared among several process invocations, or private to each process? Should the extensionalization rules depend only on intensional files' names, or also on their contents? These and other questions quickly convinced us that we could not hope to simply add code and options to IFS0 to support directories and writable files – the resulting system would have been too inflexible and arbitrary.

Instead, we arrived at a design that uses a user space kernel together with an external extensionalizer specified by the user at run time via an environment variable. IFS1 improves on IFS0 by adding user-defined escapes, and by invoking the extensionalizer only for pathnames that do not exist extensionally, and which either are unusual (i.e., end in "`!`" or contain parentheses) or contain a symbolic link component that expands to an unusual pathname. The "`!`" pathnames invoke a user-defined escape mechanism; pathnames containing parentheses are interpreted much as in IFS0.

Users define a pathname escape mechanism with the environment variable `IFS_COMMAND`. For example, we can define `IFS_COMMAND` to interpret pathnames like `F!`, where `F` does not exist but `F.Z` does. The following script copies `F` to the standard output, and caches the extensional uncompressed file in `/tmp`.

```
#!/bin/sh
- zcat <$1.Z  ||  exit 2 # ENOENT
```

The program "`-`" runs its arguments as a command, directs standard output to a temporary file in an IFS-managed directory in `/tmp`, and yields the name of the temporary file. Suppose the above script is named `/bin/zcat.ext`, and we set `IFS_COMMAND` to `/bin/zcat.ext`. Afterwards, references to `F!` force `F` to be extensionalized:

```
$ ls -l F*
-rw-r--r--  1 pre     788213 Nov 10 22:00 F.Z
$ ls -lL F
F not found
$ ls -lL F!
-r--------  0 pre    1297200 Nov 10 22:01 F!
```

The `IFS_COMMAND` program can specify further actions for IFS1 to take by outputting other information. It tells IFS1 that the new file is a temporary by prefixing its pathname with `(unlink)`; IFS1 then unlinks the file after doing the system call on it, and before returning to the user program. For example, the IFS0 abbreviation

D/(command)

is implemented in IFS1 by defining it to be equivalent to the pathname

D/$(- command)

where "`-`" is the temporary file command described above. Here is a simple implementation of the "`-`" command:

```
#!/bin/sh
echo -n "(unlink)/tmp/ifs$$"
exec "$@" >/tmp/ifs$$
```

The user-defined escape feature permits something like make processing to be performed directly by IFS1. The zcat.ext script implements the makefile rule

```
.SUFFIXES: .Z
.Z:
        uncompress $*
```

but works anywhere in the file system, handles compressed files in read-only directories, does not require explicit invocation of a special program like make, and automatically cleans up the results.

The "!" escape character is required to force IFS1 extensionalization. This convention avoids surprises; extensionalization happens only for pathnames containing explicit IFS0 intensional expressions, pathnames containing symbolic links that expand to such expressions, and pathnames ending in "!". This makes it possible for the user (and IFS1) to determine easily whether a pathname will involve extensionalization. Also, with csh this convention works well, since the command line abbreviation "!!!" means "do the previous command, but this time extensionalize its last argument".

Specifics of IFS1 Pathname Interpretation

The wrapper that IFS1 interposes around a pathname system call has no extra effect if the system call at first succeeds, or if it fails for reasons other than the file's lack of existence. However, if the call fails because of a missing file (i.e., errno=ENOENT), and if the environment variable IFS_COMMAND is set, IFS1 examines the pathname looking for parenthesized commands, or for symbolic links that expand to parenthesized commands. If if finds any, it extensionalizes them using rules similar to IFS0; otherwise, if IFS_COMMAND is nonempty and the pathname ended with "!", IFS1 invokes IFS_COMMAND, passing it the pathname (minus "!"), the name of the system call, and some extra information to tell the program whether the file is being accessed for reading or writing. If the program fails, IFS1 returns –1 and sets errno to the exit status of the program; otherwise, IFS1 retries the system call with the new pathname output by the program, and immediately unlinks the new pathname if the program preceded it with the string (unlink).

Sample IFS Applications

Previous sections have shown small examples like conditional symbolic links and date time stamps. This section shows some more elaborate examples of the IFS mechanisms in use.

FTP Storage

With IFS we can store a large file F on another site, and use FTP to retrieve the file whenever needed. For example:

```
$ ln -s '(catftp alex.sp.cs.cmu.edu'\
> ' cs-techreports/list.of.papers)' \
> internetTRs
$ ls -l internetTRs | fold -45
lrwxrwxrwx  1 stott          57 Nov 14 15:40
internetTRs -> (catftp alex.sp.cs.cmu.edu cs-
techreports/list.of.papers)
$ wc internetTRs
4638    4638  260659 internetTRs
```

where catftp is a shell script that invokes the ftp command with anonymous login.

File Fingerprinting and Encryption

Suppose we trust our local workstation, but are using networks or file servers that are less reliable, and we want to check against inadvertant or malicious modification of a file whenever we access it. We have written a program called md4cp that is like cp, but the copy that results is an intensional file that verifies the original against a fingerprint generated by the MD4 message digest algorithm [11]. Here is the source code for md4cp: it uses md4 to generate the fingerprint, and creates an intensional file that will use md4verify to check the fingerprint.

```
#!/bin/sh
if [ -d $2 ]
then F=$2/'basename $1'
else F=$2
fi
ln -s "\$(md4verify $1 'md4 <$1')" $F
```

With this we can create fingerprinted files. For example. suppose . is reliable, but /usr/local is not. We make a new self-checking X server as follows:

```
$ md4cp /usr/local/bin/X11/X .
$ ls -l X | fold -45
lrwxrwxrwx  1 stott          66 Nov 20 17:18
X -> $(md4verify /usr/local/bin/X11/X 54fbda3
4ee46771858401f2494f72b1d)
```

Before invocation, X first verifies that /usr/local/bin/X11/X still has the indicated checksum. Of course, IFS isn't necessary for this particular example: since the X server is a program, we could have just written a small shell script that verified and invoked the actual X server. But the IFS approach is more general than the shell script approach, because it works for non-executable files as well.

The source code for md4verify is as follows:

```
#!/bin/sh
if checksum='md4 <$1'
then
     case $checksum in
        $2) echo -n $1 ;;
        *)  exit 22  # EINVAL
     esac
else exit 2 # ENOENT
fi
```

It should also be possible to store a file in encrypted form using IFS; reading the file would invoke a window-based password requester that demands the password directly from the user. (The user interface of the traditional UNIX `crypt` command is inappropriate for IFS, because it disturbs the application screen layout.) This could be done with a command like `ln -s '(xcrypt F.x)' F`.

A Versioning File System

We have implemented a simple versioning file system with function reminiscent of 3DFS [8]. Our file system is based on RCS [13]. It is implemented with an extensionalizer called `rcs.ext` that has two major features: access to old versions, and automatic checkout of the latest version.

Access to Old Versions

With `rcs.ext`, the name *p*`,rn!` evaluates to a temporary file containing a copy of pathname *p*'s version *n*. For example, the shell command

```
diff ifs1.h,r1.3! ifs1.h,r1.4!
```

compares version 1.3 to version 1.4 of the file `ifs1.h`. This behaves much like the traditional RCS command

```
rcsdiff -r1.3 -r1.4 ifs1.h
```

but IFS has the advantage that old versions are available to all commands, not just those which have a special RCS wrapper program. For example,

```
grep extension ifs1.h,r1.3!
```

searches for the string `extension` in version 1.3 of `ifs1.h`. With the traditional approach, a new `rcsgrep` wrapper program would have to be written, but with IFS we can use ordinary `grep`.

Automatic Checkout

Developers often test small changes to a set of source files in a private area that contains copies of the files. In such an environment, it is convenient to have unchanged files checked out automatically when they are needed. Modern versions of `make` have automatic checkout built in; `rcs.ext` implements automatic checkout for all programs, not just `make`. If `rcs.ext` is invoked on a file *F* corresponding to an ordinary RCS file, it extensionalizes the file by invoking the RCS checkout command `co -q` *F*. The following example shows how `rcs.ext` extensionalizes the file `OK` the first time that it is referenced.

```
$ ls -l
total 2
drwxrwsr-x 3 p  512 Nov 20 14:11 RCS
lrwxrwxrwx 1 p   17 Nov 20 14:10 OK -> $(rcs.ext OK)
$ head -1 OK
Here are the detailed results of the first round:
$ ls -l
total 3
drwxrwsr-x 3 p  512 Nov 20 14:11 RCS
-r--r--r-- 1 p 1170 Nov 20 14:12 OK
```

Lazy Recursive Checkout of Directories

If `rcs.ext` is invoked on a directory *D* corresponding to a subdirectory `RCS/`*D* of the RCS hierarchy, it lazily extensionalizes *D* by `mkdir`ing it and populating it with intensional files, one for each entry in `RCS/`*D*. The following scenario shows one step of `rcs.ext`'s lazy recursive directory extensionalization prompted by a user's `cd` command.

```
$ ls -l RCS/ifs
total 21
drwxrwsr-x 2 p   512 Nov 20 14:17 doc
-r--r--r-- 2 p 18418 Oct 31 15:55 i.c,v
-r--r--r-- 2 p  1758 Oct 31 15:55 i.h,v
$ ls -l
total 2
drwxrwsr-x 4 p   512 Nov 20 14:17 RCS
lrwxrwxrwx 1 p    14 Nov 20 14:19 ifs -> $(rcs.ext ifs)
$ cd ifs
$ ls -l
total 4
lrwxrwxrwx 1 p    10 Nov 20 14:19 RCS -> ../RCS/ifs
lrwxrwxrwx 1 p    14 Nov 20 14:19 doc -> $(rcs.ext doc)
lrwxrwxrwx 1 p    16 Nov 20 14:19 i.c -> $(rcs.ext i.c)
lrwxrwxrwx 1 p    16 Nov 20 14:19 i.h -> $(rcs.ext i.h)
```

Implementation

The RCS file system extensionalizer `rcs.ext` is a `perl` script containing about fifty lines, including commentary. A listing appears in Appendix 1.

A `tar` File System

Although `tar` archive files have become a standard way to store collections of files, particularly compressed archives, they are hard to inspect. De-archiving them takes time, uses disk space, and must be cleaned up.

We can avoid archive file problems using IFS1. As an example, here is a simple `IFS_COMMAND` shell script that automatically extensionalizes `tar` files into their corresponding directories.

```
#!/bin/sh
if [ -f $1.tar ]
then = tar xf - <$1.tar
elif [ -f $1.tar.Z ]
then zcat $1.tar.Z | = tar xf -
else exit 2 # ENOENT
fi
```

With this, we can `cd` directly into `.tar` files or `.tar.Z` files. The program "`=`" is an IFS1 utility that extensionalizes a command in a temporary directory in `/tmp`, which is eventually removed by a cleanup daemon.

This approach works well for small archives. Large archives can cause problems, however, because the de-archival will store megabytes of stuff that we mostly don't care about in `/tmp`. Often there is not enough space left to do this safely.

Using the IFS1 approach, we implemented a lazy de-archiver that produces intensional files that refer back to the archive for more de-archival, if this turns out to be wanted. We did this by extending `tar` itself to support intensionality. In about 200

lines of C we added two options to Gnu `tar` 1.11.1:

- The `H` option indicates that the pathname arguments are actually numbers of specific records in the archive, each of which is the archive header record for a file to be extracted. For example, the command `tar xfH foo.tar 35 43` extracts the two files whose headers are at record numbers `35` and `43` (after verifying these headers correct). Access is fast for archives that are ordinary files (not fifos). Compressed archive access is much slower, since the entire archive must be sequentially uncompressed.
- The `I` option causes `tar` to extract files as intensional files that invoke `tar` with the `H` option, selecting the appropriate header record in the archive for extraction.

The Gnu `tar` program also provides three other useful options: `Z` announces that the input file should be uncompressed first, and `O` announces that extracted files should be routed directly to stdout. These extra options make it possible to satisfy all our de-archival needs with `tar` itself.

The scenario in Appendix 2 was obtained with the Gnu Chess tar file (which was compressed). After `cd`'ing into the compressed `tar` file, we poked around. The `chdir` system call handles the escaped pathname `gnuchess-4.0.pl60!`, causing a lazy intensional version of the `tar` file directory structure to be created in a subdirectory of `/tmp`. Although the original `tar` file is 400 Kbytes, the image in `/tmp` is only a few thousand bytes. It consists of the directories in the original and intensional files pointing at their header records into the `tar` file. E.g., `gnuchess.book` is an intensional file equivalent to

```
(tar xfHOZ /tmp/gnuchess-4.0.pl60.tar.Z 334)
```

which announces: to get this file, extract from the original compressed archive, putting the file with header record 334 to stdout. Since any dynamically linked program automatically incorporates IFS1, `ls`, `head`, and `diff` all get the files you would expect.

The lazy `tar` extensionalizer is as follows:

```
#!/bin/sh
if [ -f $1.tar ]
then = tar xI 'realpath $1.tar'
elif [ -f $1.tar.Z ]
then = tar xIZ 'realpath $1.tar.Z'
else exit 2 # ENOENT
fi
```

This is much like the eager script shown earlier, except that provision is made to save the full pathname of the original tar file, since it may need to be referred to later. The `realpath` utility produces an absolute pathname from a given pathname. The key feature is the new `I` option of `tar`: when run in the extensional directory built by "=", it produces the intensional files seen in Appendix 2.

The lazy approach taken to `tar` navigation taken here is space efficient, but of course charges a high price in performance. The response time for compressed archives can be irritatingly slow, since the archive is uncompressed and scanned sequentially on each access. However, it would not be difficult to change the extensionalizer here to check how much space is left in `/tmp`, and use lazy extensionalization only if this is necessary.

Design Issues for Intensional File Systems

Basic Design Principles

Our design principles for IFS were influenced by those of Satyanarayanan [12], and included the following:

- *Clients have cycles to burn.*
  More and more, file client processors are nearly as fast as (and a much less scarce resource than) file server processors.
- *Cache whenever possible.*
  Like CPU cycles, client caches are becoming relatively abundant and parallelism is maximized by moving as much work as possible onto the clients.
- *Let users exploit usage properties.*
  The people who actually use intensional files are more likely to know their usage properties than even the best file system designer. Keeping the IFS system design simple makes it easier for users to gain intuition about how to set up their intensional files efficiently.
- *Minimize system-wide knowledge and change.*
  The more local an intensional file's specification is, the easier it is to build and scale the file system containing that file, particularly to large distributed systems.
- *Minimize security risk.*
  A primary danger of intensional files is that they invite abuse. Malicious users could replace extensional user files with intensional files that log keystrokes or perform other unspeakable acts, for example. This danger cannot entirely be eliminated; it is analogous to the problem of letting users set their `PATH` environment variable to malicious users' directories. However, the danger should not be unnecessarily enhanced, or made so subtle that users will not be able to avoid it easily.
- *Have adequate performance.*
  File systems must provide adequate performance, or they will not be used. Of particular importance is *interactive* performance.

Since IFS0 and IFS1 are just first prototypes, high performance is not essential; but enough is needed to get adequate hands-on experience.

Introduction of Intensional Files

The first design issue in implementing intensional files is how the user is supposed to conceptualize them. Intensional files must be mapped into the existing file system model in a coherent way.

Even IFS's modest implementation exposes several fundamental design considerations for intensional file systems, including the following:

- *How can we add intensional files gracefully?* It should be easy for users to see exactly which files are intensional, and what rules are being used to extensionalize them. A principal goal of our work was to come up with simple, intuitive rules governing intensional files. This is in addition to the obvious goal that unmodified traditional programs should transparently access the extensional counterparts of intensional files.
- *How heavyweight should intensional file systems be?* Traditional UNIX file systems contain a large number of files and directories, and (for security reasons) can be manipulated only by the superuser. At first, we followed this model, but eventually we rejected it because it is inappropriate for a design meant for personal use and tailoring.
- *Should extensionalization be done by the file system client or by the server?* Doing the work on the server means that we don't have to touch the clients; but doing it on the clients increases parallelism and decreases network overhead.

These issues are not new to intensional files. The introduction of symbolic links in Berkeley UNIX raised most of these issues, albeit in somewhat simpler ways. Symbolic links are an important special case of intensional files: the intensional counterpart of a symbolic link is the name of the linked-to file, and the extensional counterpart is the contents of the linked-to file. Once we saw that symbolic links were special cases of intensional files, it became obvious that one way to implement intensional files is to generalize symbolic links.

A more radical analogy between intensional and traditional file systems is that a traditional file system can be modeled as an IFS backed by a traditional UNIX special device. Thus, modern microkernels with file systems migrated out of the kernel have much in common with intensional file systems.

Related Work in File Systems

There has been an enormous amount of interest recently in extending file systems to support new functionality, as indicated by the recent USENIX workshop on file systems [1]. New approaches strike a compromise between flexibility, performance, and completeness. For example, operating systems like Mach permit the entire file system to reside outside the kernel, yet few users would be capable of writing a new Mach file system or even making ad hoc file system modifications. UNIX System V Streams [2] are much less flexible, providing a service that is user-programmable but not really user-extensible. (An extensible variant [10] has been proposed, however.)

Stackable file systems like Ficus [6, 9] are a promising way to implement new file system concepts atop traditional systems. They provide a disciplined scheme for incorporating features into a file system by modeling the file system design as a protocol stack, and letting one insert new protocols onto the stack. This can greatly ease implementation by file system experts, but it's not a task for novices, and it requires root permissions.

One general approach that has been proposed for extensible file systems is *file system interface mapping*, in which basic system calls like `open` and `read` can be replaced by user code. This approach can encourage exploration, but a key issue is how the mapped interface is specified by the user.

One approach for file system interface mapping allows users to associate a process with a given file that implements the appropriate mapping. The kernel is modified to route system calls for the file to this process. Bershad and Pinkerton [3] describe *watchdogs*, which are user-defined processes that the user can attach to a file. Watchdogs are notified about each system call affecting the file. They have been used to implement access control, file compaction, mail biffing, directory views, transparent remote file access, etc. Welch and Ousterhout [14] propose *pseudo-devices*, basically processes that can be treated like files, again by interface mapping. Various sticky programming jobs such as interacting with device drivers, X servers, and TCP/IP services can be simplified by treating them as pseudo-devices.

Another general approach is to encourage user redefinition of the basic system call interface in `libc`. Any such redefinitions are automatically adopted by all dynamically linked programs. This approach requires no kernel modifications, and works entirely in user space. COLA [8] is a system call interception scheme developed at Bell Labs. Basically, COLA is to system calls at the user level as Ficus is to file system protocols at the kernel level. With COLA, a programmer can insert a new definition of, say, the `read` system call; the programmer-inserted code operates as a layer between the actual system calls and the virtual `read` call that is visible to the rest of the program. We had independently developed techniques similar to COLA in a memory management profiler, and were thus familiar with its implementation strategy. It is the approach used in IFS.

IFS1 Implementation

IFS1 is currently implemented atop SunOS 4.1.x, and operates by intercepting the few system calls that refer to pathnames, notably `open` and `stat`. By manipulating an environment variable, a user can tell the dynamic linker to preload IFS1 before linking the standard C library; the `IFS_COMMAND` environment variable enables and disables IFS1 itself.

Aside from a few subtle issues about which system calls to intercept, and how to prevent interception self-loops, IFS1 itself is surprisingly simple: its kernel is only 700 lines of C code. Although there are some obvious ways to improve performance, little attempt has been made to tune the code.

IFS1 Performance

Despite this rather academic attitude towards performance, IFS1 is useful in practice. In ordinary use it's not even noticed. For example, starting up a new `csh` ordinarily takes about 500 system calls in our environment; IFS1 adds just seven system calls to this total, five to dynamically prelink the IFS1 library, and two to check whether `~/.history` (a nonexistent file) is intensional. On a Sparcstation ELC running SunOS 4.1.2, IFS1 adds only 32 μs overhead to each pathname system call in the usual case of an existing extensional file. There is no overhead for system calls like `read` and `write` that do not interpret pathnames. Since intensional files are cached in `/tmp`, the normal SunOS `tmpfs` cacheing means that one can even construct examples where IFS1 is faster than ordinary file system access. Another situation where IFS1 compares favorably with conventional file systems is the not unusual combination of a slow remote file server, a fast client, and large local extensionalized files.

The key reason for IFS1's speed is that it acts only in unusual circumstances, when a pathname does not exist, and has an unusual form or a symbolic link. When IFS1 does not act, the overhead is low – generates no extra system calls in the usual case. When IFS1 does act, the overhead of the IFS1 mechanism itself is low even with the current inefficient prototype; under a tenth of a second on a Sparcstation ELC as measured by `csh time`. This is perfectly adequate for interactive applications.

Security

IFS1 takes a naive approach to security. The process that extensionalizes a pathname has almost all the privileges of the process that is accessing the pathname. However, all the extensionalizer's file descriptors are closed except for standard output; this exception is more to avoid inadvertent side effects than to close security holes. Also, for safety's sake, the dynamic linker does not prelink IFS1 in setuid programs.

IFS uses only facilities already available to ordinary users, and does not require superuser access either for installation or use, so in some sense IFS1 does not introduce new security issues. In ordinary use, we expect that avoiding Trojan horse extensionalizers will be about as difficult as avoiding ordinary Trojan horse programs in one's `PATH`. Also, as the MD4 example showed, IFS can be used to improve security in an insecure environment.

Future Work

We consider IFS0 and IFS1 as successful experiments. Their success suggests some obvious extensions for the next version.

- *Writeable intensional files*
  One would like the ability to not only read intensional files, but also write them. For example, writing to the extensional counterpart of a compressed file should cause the `compress` program to be invoked to store the actual file on disk. Even more than intensional directories, this requires careful resource allocation; for example, we must be careful that inadvertent process death does not undo the effects of `write`.
- *Security*
  IFS1's naive approach to security is simple and easy to explain, which is an important virtue in security matters, but some potential users have expressed fears about bugaboos like Trojan horse extensionalizers. IFS1 could be extended to run extensionalizers in a more restricted environment.
- *Cacheing*
  IFS does little cacheing of extensionalizations of files: even if the same process opens a file twice, IFS dutifully does the full work of extensionalizing the file twice. The "`-`" and "`=`" utilities could be made much more intelligent. Although extensional caches would improve performance, they would come at the cost of complicating the implementation with the usual questions of cache management.
- *Managing imaginary file systems*
  Some tools need directory information that may not exist in an intensional system. For example, consider the use of `du` on an intensional directory, which would ordinarily discover how much storage the (extensional version of the) directory occupies on disk. Should IFS materialize the entire directory just so that `du` can see how big the result is? Such questions show there is lots of work to do here.
- *When is it worthwhile to make files intensional?*
  It is not normally worth making the executable version of a large software system an intensional file, since its materialization can

be costly. Currently it seems that the main payoff of intensionality comes with files that are relatively easy to materialize, or that are relatively large, or both.

Conclusion

It is high time to open file systems up to direct user manipulation, rather than leaving them closed systems that only qualified wizards are permitted to design. Mundane tasks like versioning, encryption, and compression should be handled by a file system directly, instead of by the current multitude of ad hoc approaches, one per application. Current dynamic linking technology (and the object-oriented technology of the not-too-distant future) should make it easy for users to add new file naming conventions in a systematic way.

IFS builds on the idea of intensionality to give users a flexible way to extend their file systems' naming conventions. The examples in this paper show that intensionality can replace and extend many ad hoc file system techniques.

Over the past year we have tried to isolate a part of intensionality that can form a practical basis for file systems development. The current design of IFS1 reflects both a basic "expressions as names" mechanism and a "name escape" mechanism. We hope further experience with IFS will produce a important mechanism for day-to-day use. If nothing more, however, IFS represents the kind of file system interface that lets a hundred flowers blossom and a hundred schools of thought contend.

References

1. *Proceedings of the First USENIX File Systems Workshop*, Ann Arbor, MI, May 1992. P. Honeyman (program chair).
2. AT&T, *UNIX System V Release 3.2 Streams Programmers Guide*, Prentice-Hall, 1989.
3. Bershad, Brian N. and C. Brian Pinkerton, "Watchdogs – Extending the UNIX File System," *Computing Systems*, vol. 1, no. 2, pp. 169-188, 1988.
4. Eggert, Paul R. and D. Stott Parker, "An Intensional File System," *Proc. First USENIX File Systems Workshop*, pp. 145-146, Ann Arbor, May 1992.
5. Gifford, D.K., P. Jouvelot, M.A. Sheldon, and J.W. O'Toole Jr, "Semantic File Systems," *Proc. of the 13th ACM Symposium on Operating Systems Principles*, pp. 16-25, October 1991.
6. Heidemann, J.S. and G.J. Popek, "An Extensible, Stackable Interface for File System Development," Technical Report CSD-900044, UCLA Computer Science Department, Los Angeles CA 90024-1596, December 1990.
7. Hendricks, D., "A Filesystem for Software Development," *Proc. USENIX Summer Conference*, pp. 333-340, Anaheim, June 1990.
8. Korn, D. and E. Krell, "The 3-D File System," *Proc. USENIX Summer Conference*, pp. 147-156, Baltimore, June 1989.
9. Page, T. and R. Guy, "The Ficus Scalable File System," *IEEE TCOS Newsletter*, vol. 5, no. 3, pp. 19-20, IEEE Computer Society Technical Committee on Operating Systems and Application Environments, Fall 1991.
10. Rees, Jim, Margaret Olson, and J. Sasidhar, "A Dynamically Extensible Streams Implementation," *USENIX Conference Proceedings*, pp. 199-207, Phoenix, AZ, Summer 1987.
11. Rivest, R., "The MD4 message digest algorithm," RFC 1186, Network Working Group, October 1990.
12. Satyanarayanan, M., "The Influence of Scale on Distributed File System Design," *IEEE Transactions on Software Engineering*, vol. 18, no. 1, pp. 1-8, January 1992.
13. Tichy, Walter F., "RCS – a system for version control," *Software—Practice & Experience*, vol. 15, no. 7, pp. 637-654, July 1985.
14. Welch, Brent B. and John K. Ousterhout, "Pseudo-Devices: User-Level Extensions to the Sprite File System," *USENIX Conference Proceedings*, pp. 37-49, San Francisco, Summer 1988.
15. Zalta, E.N., *Intensional logic and the metaphysics of intentionality*, MIT Press, Cambridge, MA, 1988.

Author Information

Paul Eggert is Director of R&D at Twin Sun, Inc. He got his Ph.D. in Computer Science at UCLA in 1981, and is currently working on CASE tools, groupware, and software reliability. He can be reached as `eggert@twinsun.com`, or by U.S. mail at Twin Sun, Inc., 360 N. Sepulveda Blvd., El Segundo, CA 90245.

Stott Parker is Professor in the Computer Science Department at UCLA. He joined the department in 1979 after getting his Ph.D. at the University of Illinois. Contact him at `stott@cs.ucla.edu`, or by U.S. mail at Computer Science Department, University of California, Los Angeles, CA 90024-1596.

Appendix 1: RCS Intensional File System Source Code

```
#!/usr/local/bin/perl

# IRCS extensionalizer
#   $Id: IFS.usenix,v 1.3 1992/11/22 04:06:21 eggert Exp eggert $

# Disable IFS internally.
delete $ENV{'IFS_COMMAND'};

$pathname = @ARGV[0];

if ($pathname =~ /^(.*),r([^$,\/:;@]*)$/) {
    # Map "foo,rN" to temporary containing revision N of file foo.
    $tmp = '/tmp/ifs'.getppid;
    exit if
        (print "(unlink)$tmp")
        && open(STDOUT, ">$tmp")
        && (system 'co', '-q', "-p$2", $1) == 0;

} elsif (print $pathname) {

    if ($pathname =~ /^(.*\/)([^\/]*)$/) {
        $dirl = $1;
        $file = $2;
    } else {
        $dirl = '';
        $file = $pathname;
    }

    if (! -d "${dirl}RCS/$file") {
        # Map to latest revision on default branch.
        exit if (system 'co', '-q', $pathname) == 0;

    } elsif (opendir(DIR, "${dirl}RCS/$file")) {
        # Map to corresponding directory.

        # Unlink any symbolic link; it's probably what invoked us.
        unlink($pathname) if -l $pathname;

        # Then create dir and subsidiary symlinks.  */
        if (mkdir($pathname, -1) && symlink("../RCS/$file", "$pathname/RCS")) {
            $ok = 1;
            for (readdir(DIR)) {
                if ($_ ne '.' && $_ ne '..') {
                    s/,v$//;
                    $ok &= symlink("\$(rcs.ext $_)", "$pathname/$_");
                }
            }
            exit if $ok;
        }
    }
}

exit 2 # ENOENT
```

Appendix 2: Intensional tar File System Scenario

```
$ ls -l gnuchess-4.0.pl60.tar.Z
-rw-------  1 stott      416029 Nov 16 20:03 gnuchess-4.0.pl60.tar.Z
$ cd gnuchess-4.0.pl60!
$ ls
gnuchess-4.0.pl60/
$ cd gnuchess-4.0.pl60
$ ls -l
total 2
drwx--x--x  2 stott         528 Nov 16 20:13 doc/
drwx--x--x  2 stott         168 Nov 16 20:13 misc/
drwx--x--x  2 stott        1120 Nov 16 20:13 src/
drwx--x--x  2 stott         444 Nov 16 20:13 test/
$ du .
1       ./doc
0       ./misc
1       ./src
0       ./test
2       .
$ ls -l misc | cut -c1-12,46-
total 0
lrwxrwxrwx  ChessFont -> (tar xfHOZ /tmp/gnuchess-4.0.pl60.tar.Z 260)
lrwxrwxrwx  gnuchess.book -> (tar xfHOZ/tmp/gnuchess-4.0.pl60.tar.Z 334)
lrwxrwxrwx  gnuchess.lang -> (tar xfHOZ/tmp/gnuchess-4.0.pl60.tar.Z 319)
lrwxrwxrwx  gnuchess.nunn.book.Z -> (tar xfHOZ /tmp/gnuchess-4.0.pl60.tar.Z 426)
lrwxrwxrwx  match -> (tar xfHOZ /tmp/gnuchess-4.0.pl60.tar.Z 316)
$ head -7 misc/gnuchess.book
!
! Opening Library for CHESS
!
!  Copyright (c) 1988,1989,1990 John Stanback
!  Copyright (c) 1992 Free Software Foundation
!
!  This file is part of GNU CHESS.
$ cd src
$ ls
Makefile@       debug12.h@      dspcom.c@       gdxbkstats.c@   nuxdsp.c@
Makefile~@      debug13.h@      eval.c@         genmoves.c@     postprint.c@
README@         debug16.h@      extern.h@       gnuan.c@        search.c@
ataks.c@        debug256.h@     game.c@         gnuchess@       systems.h@
ataks.h@        debug4.h@       gbookdist.c@    gnuchess.h@     util.c@
bincheckr.c@    debug40.h@      gdbm.h@         init.c@         uxdsp.c@
book.c@         debug41.h@      gdbmbkstats.c@  main.c@         version.h@
checkbook.c@    debug512.h@     gdbmconst.h@    match@
checkgame.c@    debug64.h@      gdbmdefs.h@     membkstats.c@
debug10.h@      debug8.h@       gdbmerrno.h@    nondsp.c@
$ ls -lL *book*
-rw-------  1 stott       42755 Nov 16 20:24 book.c
-rw-------  1 stott       21787 Nov 16 20:24 checkbook.c
-rw-------  1 stott        1963 Nov 16 20:24 gbookdist.c
$ diff Makefile*
39c39
< DISTDIR=/tmp_mnt/home/fsf/cracraft/gnuchess-4.0
---
> DISTDIR=/tmp_mnt/home/fsf/cracraft/g4p60
$ pwd
/tmp/.ifs/7763/7802304/gnuchess-4.0.pl60/src
$
```

UNIX Kernel Support for OLTP Performance

Hyuck Yoo & Tom Rogers – Sun Microsystems, Inc.

ABSTRACT

UNIX machines are increasingly being used for Online Transaction Processing (OLTP) in database applications. There have been several mismatches between OLTP requirements and UNIX kernel facilities necessary to implement them that have led to performance bottlenecks. In this paper we describe two kernel features that improve OLTP performance on UNIX. We describe enhancements to the virtual memory system and I/O system in the UNIX kernel, and evaluate the performance of the new kernels with a database benchmark. The results show that the enhancements achieve significant improvement in OLTP performance.

Introduction

Traditionally, OLTP in database applications has been centered around mainframes and their proprietary operating systems because these machines were the only ones powerful enough for database applications. With the increased power of workstations and the popularity of the client-server computing model, UNIX machines have become attractive for OLTP applications. It is expected that a recent trend of "rightsizing" will lead to more OLTP applications running on UNIX machines.

Transactions in typical OLTP environments such as banking and reservation systems are simple and of short duration (less than a second), and need quick response time. OLTP performance is a function of three main factors: underlying hardware, operating system (OS), and DBMS software[1]. In addition, OLTP performance can be affected by many other variables such as system configuration and tuning parameters, and it involves many issues such as transaction processing monitors. We do not attempt to address all the problems that OLTP poses. Rather, we address the OS issues which are related to UNIX kernel.

An OLTP workload can be characterized as having

- intensive disk I/O
- a large number of concurrent users
- a high degree of sharing between users

These are quite different requirements from those to which the original UNIX and its variants were designed, i.e., (relatively) small scale time-sharing systems with little sharing between users. So it is not surprising for UNIX to run into many limitations in handling OLTP applications efficiently [5]. Since [5] was written, much has changed in UNIX as well as RDBMSs.

However, there is very little published literature on UNIX and OLTP performance. [6] deals with mostly hardware issues such as system bus and cache mechanisms in shared memory multiprocessor systems. [18] reports the cache and translation look-aside buffer misses of several applications including an OLTP application on RISC architecture systems. [9] and [10] focus on new transaction models and transaction architectures. [8] describes a library implementation of transaction facilities, whereas the OLTP environment considered in this paper assumes that the RDBMS exists as a separate entity on top of UNIX, not as a library. None of these references focuses upon UNIX kernel performance issues.

The Database Engineering group at Sun has been investigating UNIX and OLTP performance with various RDBMSs. In this paper, we present the findings based on our experience, and describe how we improved OLTP performance using our UNIX kernel called the *Sun Database Excelerator* (SunDBE).

The rest of the paper is organized as follows: Section 2 introduces mismatches between UNIX and RDBMS, and points out the fundamental services from the UNIX kernel for RDBMSs. Section 3 gives a description of our virtual memory (VM) enhancement and Section 4 describes how we improved disk I/O performance. The experiments based on the ideas presented in Sections 3 and 4 are reported in Section 5. Section 6 discusses some issues to be investigated and Section 7 concludes the paper.

Mismatches between UNIX and RDBMS

There are many mismatches between UNIX kernel facilities and RDBMS OLTP requirements. Two main examples are the file system and the scheduler. Most RDBMSs do not use the file system due to several reasons. The first reason is speed.

[1] By DBMS, we mean contemporary, full-function relational DBMS (RDBMS), not old flat-file based systems.

When a large number of I/Os are taking place, read() and write() calls on file systems take too much time just to copy the buffer in and out of kernel virtual memory. Portability is another reason to avoid the file system. The overhead of read() and write() on files can be minimized by mmap(). So the speed problem is solved by mmap(), but the mmap() call is not used because it is available only on certain platforms. RDBMSs do not want calls that are not portable. Another reason not to use the file system involves reliability. An RDBMS has to secure the data regardless of unlikely events such as a power failure. When data buffers are maintained by the UNIX kernel, there is no guarantee that during a crash a certain data record has been pushed to disk. Therefore, RDBMSs use raw disks and maintain the buffer cache.

Another example of a mismatch besides the file system is the UNIX scheduler. Suppose that the UNIX scheduler preempts a process holding a critical resource such as a latch[2] in the RDBMS log. Other processes that want the latch have to wait for the holder to be rescheduled. Typically, the preempted process will be placed in the end of the queue (by round robin policy). So although the next process in the run queue is scheduled to run, it can't run because the latch is already taken. So the processes waiting for the latch will waste their scheduling turns. In this case, the best thing is not to preempt the process holding a latch. In summary, preemptive scheduling could be problematic to OLTP. Since every transaction involves I/Os during execution, it may be a better scheduling policy to hold the preemption until the transaction voluntarily gives up the CPU for an I/O request.

As an attempt to overcome the mismatches, modern RDBMSs try to minimize their dependency on UNIX, especially by avoiding UNIX system calls as much as possible. However, despite wanting to bypass or eliminate most UNIX dependencies, RDBMSs still require UNIX support for the following areas:

- virtual memory to support a large number of users
- high performance I/O
- multiprocessor (MP) support

RDBMS architecture determines how OLTP users are represented in the system. There are two main approaches to RDBMS architecture: conventional and multithreaded. In this paper, we assume the conventional RDBMS, i.e., an OLTP user is represented as a (heavyweight) UNIX process. A detailed discussion of RDBMS architecture is beyond the scope of this paper.

The next two sections describe how to enhance the VM system for a large number of user processes and to provide high performance I/O in order to improve OLTP performance. Some MP support issues are discussed later.

[2]Latch is a form of mutex.

Virtual Memory

Data sharing between users in OLTP requires some form of interprocess communication facility. Most RDBMSs on UNIX extensively use the system V shared memory calls such as shmget(2), shmat(2), shmctl(2), and shmdt(2) [2]. A shared memory segment is created by shmget(), and the data buffers used in OLTP are allocated from this shared memory. User processes access the data buffers by attaching the shared memory to their address space through calling shmat(). Thus, any change made by a process is immediately visible to other processes because they are accessing the same pages. The shared memory is detached from the address space by calling shmdt(). Shmctl() is used to do various control functions such as removing the shared memory, etc. Obviously, these calls are closely related to the VM system.

Abstractions used in the SunOS VM system are address space, segment, page, and hardware address translation layer. The details of the SunOS VM system are explained in [1] and [3]. Address spaces in SunOS consist of mappings to VM objects. The most common VM object is a file. Another object is anonymous memory which does not have the name of backing store exposed to userland[3]. Each mapping is represented by a segment, and the segment driver implements the semantics of a specific segment. The mapping of both objects (file and anonymous memory) are represented by a vnode segment (segvn) [4]. The segvn driver supports both private and shared mappings.

Since shared memory is almost a file [1], when a shared memory is created, the corresponding swap space is reserved. When a shared memory is subsequently attached, the segvn driver is installed for the shared memory segment in the user process. Obviously, shared memory is a shared mapping so that physical pages in the mapping are shared between processes attaching the shared memory. However, this means that VM resources such as the page tables are allocated for each process attaching the shared memory.

When a large number of OLTP processes are running, segvn will allocate a large number of page tables. Suppose that the size of shared memory is 128M. The size of the page tables for this shared memory is about 130K in systems based on the SPARC Reference MMU (SRMMU) with a 4K page size [16]. If 100 users are running on the system, 13 MB physical memory needs to be reserved just for

[3]Userland means user address space in contrast to kernel address space.

page tables. The actual physical memory requirement is higher because of the kernel data structures associated with page tables. Note that the page tables are not pageable nor swappable. Without enough page tables allocated at boot time, the VM system has to steal the page tables from other processes when it encounters the shortage of page tables. The page table stealing is very expensive because all the pages mapped in the page table have to be flushed before being given to another process. Thus, page table stealing slows down performance substantially.

To overcome this performance bottleneck, we have implemented shared page tables (SPT) in order to support a large number of OLTP processes without allocating a large amount of page tables. The idea is simple: Create one set of page tables for a shared memory segment, and share the page tables as well as the shared memory itself among user processes. We created a new segment type (segshm) for the shared memory, and the new segment driver is installed when the shared memory is attached.

When a shared memory is created, one level 1 page table (L1PT) is allocated for the shared memory. The L1PT of the shared memory is filled with the addresses of the pages belonging to the shared memory. Fig. 1 depicts how the page tables are organized to map the pages in an SRMMU-based system. The L1PT is pointed by an SPT pointer. L2PT and L3PT stand for level 2 and level 3 page tables, respectively.

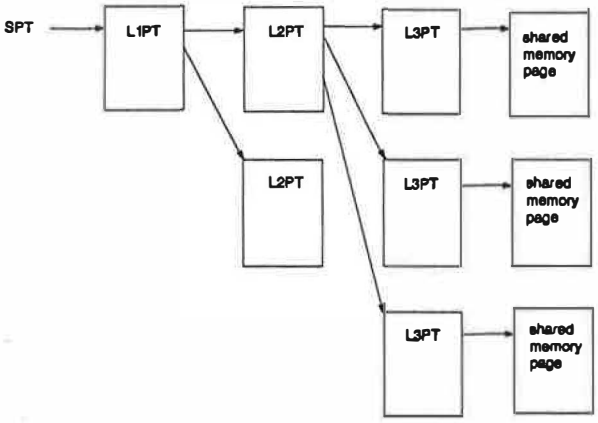


Figure 1: Page tables after shmget()

When a process attaches the shared memory, the L1PT of the shared memory is copied to the attach address of L1PT for that process. Then references to any of the shared memory from the process attaching to the shared memory is through the same page tables that the L1PT of the shared memory is pointing to. Note that the unit of sharing is an entry in L1PT, and this constrains the attach address to the size which a L2PT maps. With the shared page table, only one set of page tables is needed to support an arbitrary number of processes. Fig. 2 shows how a user process attaches the shared memory by sharing page tables in SPT. USR stands for a user process. Note that L1PT(*) and L2PT(*) are level 1 and level 2 page tables that belong to USR.

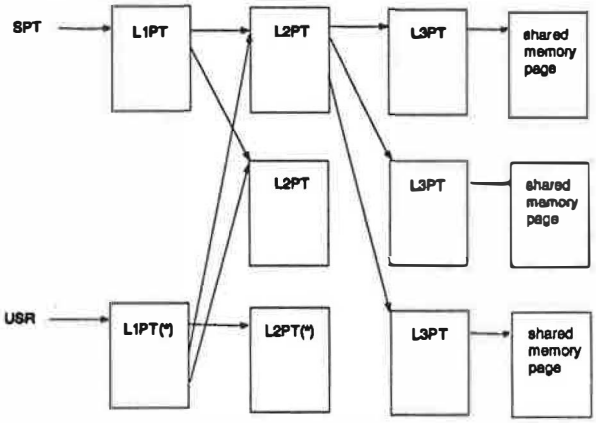


Figure 2: Page tables after shmat()

Upon shmdt(), the SPT entries in the caller's L1PT are deleted, and the caller process is separated from SPT. When the shared memory is destroyed by shmctl(), the mapping in SPT is destroyed and the page tables are freed. The pages belonging to the shared memory are also freed.

High performance I/O

Due to intensive I/O, OLTP requires a high performance I/O system that can handle hundreds of active I/O requests per second. A common approach in RDBMSs is to use I/O processes[4] to achieve desirable I/O performance. I/O processes receive I/O requests from user processes and implement the requests. Thus, user processes are decoupled from the I/O overhead. This is a simple and easy solution to achieve high performance. Since the traditional I/O facility in UNIX is synchronous I/O (SIO), an I/O process goes into the kernel and blocks until the I/O operation is finished. Thus, the number of I/O processes has to grow in order to keep up with increasing numbers of I/O requests. The disadvantage to this solution is that the large numbers of I/O processes compete with the user processes for both memory and kernel resources. Overall, performance is hurt considerably. In addition, the I/O processes need a way of synchronizing between themselves which adds still more overhead.

An alternative is to use asynchronous I/O (AIO) for I/O requests. AIO means that control returns to the caller without waiting in the kernel for the I/O request to be completed. With AIO, one or a few I/O processes are enough to keep up with the

[4] A variation of I/O processes is called disk processes. An I/O process can handle I/O requests to any disk but a disk process is dedicated to a particular disk.

large number of I/O requests. Alternatively, user processes can directly issue I/O requests (through AIO) without I/O processes. When the I/O is done, a signal is sent to the I/O requester, or the requester can mask the signal and poll the completed I/O.

Another advantage of AIO is that it avoids context switching when waiting for the I/O completion in the kernel. Since the typical transactions are of short duration and many transactions are running per second, the reduction in the number of context switches is very important in OLTP performance. For example, a typical TPC-B transaction[5] issues about 3 random I/O calls[6]. If 200 transactions are executing per second, this means that 600 context switches take place per second just for I/Os when AIO is not used.

The idea of asynchronous I/O is not new. Proprietary systems such as IBM MVS and DEC VMS have a similar facility to provide a high performance I/O system. Also, several vendors have various kinds of AIO implementations in UNIX. For example, Pyramid [11] has implemented an ioctl() interface for AIO. However, we have not seen any literature claim or disclaim the advantage of AIO over SIO in real applications. [12] describes an AIO implementation on IBM AIX based on a draft of POSIX P1003.4 standard without experiments.

SunOS provides asynchronous I/O calls such as aioread(), aiowrite(), aiowait(), and aiocancel(). As completion notification, the SIGIO signal is delivered and/or a process calls aiowait() to dequeue a request from the queue. Note that SIGIO can be delivered for more than one completed I/O request and aiowait() dequeues one request each time it is called.

There are (at least) two ways of implementing AIO in UNIX. One is to use a set of kernel processes (or threads) to handle each of the I/O requests. This approach has several advantages: 1) no change is required in device drivers or other parts of the kernel, 2) it can support any type of file, 3) the implementation is relatively easy. Another way is to "fastpath" the requests to the device driver. This implementation has several disadvantages. Mainly, it is quite difficult to support arbitrary file types because vnode operations are assumed to be performed in the context of calling process [14, 15]. Nevertheless, a big advantage of the fastpath AIO is its efficiency.

[5]This will be discussed in the following section.

[6]This is the number of the physical I/Os. The TPC-B specification has 6 logical I/O calls (reads and writes) per transaction. By the intelligent caching and LRU strategy in the I/O process, the number of physical I/Os could be much less than that of the logical I/Os. However, the actual number of physical I/Os varies on RDBMSs.

We implemented a fastpath AIO to speed up raw disk I/O performance. For aioread() and aiowrite(), SunDBE does the following:

1. record the request
2. find the proper strategy routine for the given request
3. lock the buffer
4. call the strategy routine
5. return to the caller.

When an I/O is completed, the buffer pointer is recorded in the completion queue. The entries in the queue are taken out by aiowait() calls.

Experiments

There are innumerable factors involved in determining RDBMS performance, including the hardware platform as well as RDBMS and system software. There are also many different ways of implementing an RDBMS benchmark. Notable historical examples are TP1, DebitCredit, and Wisconsin benchmarks [17]. The main problem with these benchmarks was that results obtained with them were not directly comparable to each other. To avoid this confusion, the Transaction Processing Performance Council (TPC) published three industry standard DBMS benchmarks: TPC-A, TPC-B, and TPC-C. In general, the TPC benchmarks are difficult to set up, and there are several severe restrictions to follow, including scaling of database and response time constraints. However, the TPC benchmarks do exercise the VM and I/O system (among others) heavily that we are investigating in this paper.

The benchmark that we used is similar to the TPC-B benchmark. The full details of the TPC-B benchmark specification can be found in [7]. It is basically a write-intensive benchmark that simulates a hypothetical bank. A transaction in the benchmark reads an account, updates the account, and propagates the update to the teller and branch balances. There is no rollback during the transaction execution. In our experiments for this paper, we didn't follow all the specifications in TPC-B benchmark, e.g., mirroring. Rather, we focused on identifying the performance differences solely coming from kernel changes. All the RDBMS and kernel tuning parameters remained constant while we ran the benchmark with different kernels. The performance was measured as transactions per second (TPS).

The database was scaled to 120 TPS and it was distributed across 12 disks. This was large enough to provide more realistic OLTP results, in contrast to other experiments on smaller systems such as [8] which used a 10 TPS database on one disk. We used a Sun 600 series MP machine with 128M memory to run the benchmark. The shared memory size used in the RDBMS was 60M bytes. Eight I/O processes were running in the experiments that did not use AIO.

The architecture of the RDBMS used in the experiments is shown in Fig. 3. The user processes execute the transactions, and the I/O process pushes the dirty pages in shared memory to the disks. The benchmark consists of 10 minutes of ramp-up to fill the shared memory buffers, 10 minutes steady-state run followed by 10 minutes ramp-down. TPS was measured during the steady-state period.

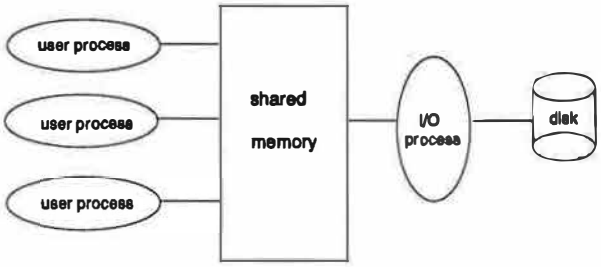


Figure 3: Architecture of RDBMS

Figure 4 shows the effect of shared page tables. Graph 1 is obtained from the SunOS 4.1.2 kernel (call it GENERIC). As more users are running, graph 1 shows that performance drops sharply. We found that this occurs when page table stealing takes place. SunOS has a variable called *npts* which can be patched, and *npts* determines the size of page tables allocated in the kernel [13]. Graph 2 is with SunOS 4.1.2 with *npts* = 0x8000 (let's call it GENERIC+). In contrast to graph 1, the performance of GENERIC+ is sustained with a small degree of degradation. Graph 3 is the result of the SunDBE kernel that uses shared page tables. At the peak, SunDBE performs 17% better than GENERIC+ and 69% better than GENERIC. The reason why SunDBE outperforms GENERIC+ is that SunDBE locks down less memory for page tables. As the number of users increases, GENERIC+ will eventually run into the lack of page tables.

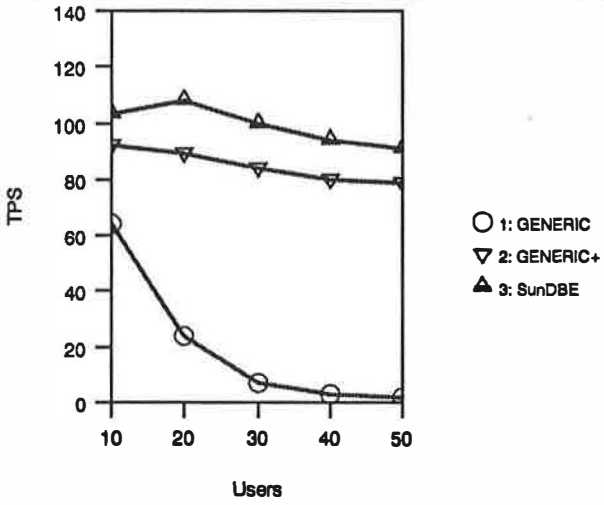


Figure 4: Effect of shared page tables

The results of using AIO are shown in Figures 5 and 6. Graph 1 of Figure 5 is GENERIC+ and graph 2 is GENERIC+ with AIO. AIO gives 3% performance increase to GENERIC+ in the peak. The gain by AIO increases as more users run and it is 19% at 50 users. This increase is due to the fact that AIO has less overhead than SIO per request, and the saving increases as more users are running. Figure 6 shows the result of using SunDBE with AIO. The peak improvement is 10%, and at 50 users AIO gives 27% increase compared with SIO. Note that AIO in Figures 5 and 6 can provide almost

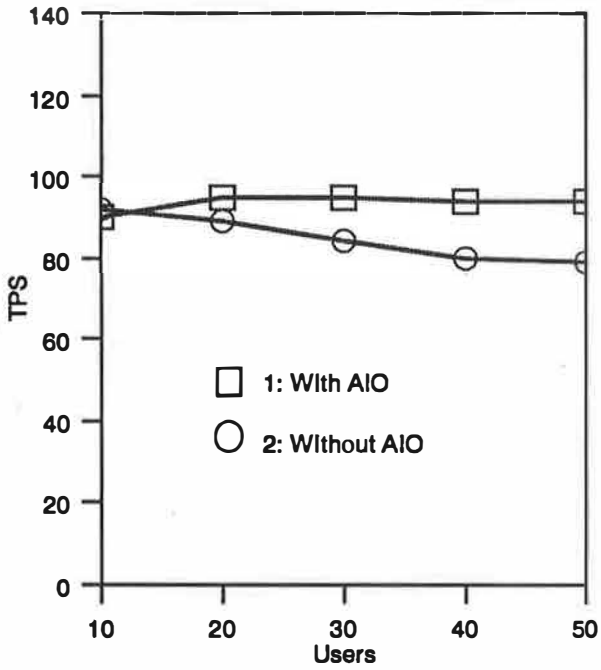


Figure 5: AIO and GENERIC+

flat performance over a range of workloads, e.g., 10 through 50 users. It is interesting to see that the performance improvement with AIO is higher with SunDBE than with GENERIC+. We believe that shared page tables in SunDBE enable more transactions to run than GENERIC+ so that the saving from AIO is greater.

Discussion

A natural question would be whether this special kernel for RDBMSs helps or harms "regular" applications. Obviously, if the applications do not use shared memory or AIO, then the kernel will provide the same functionality and same performance. However, we haven't done extensive measurements to back this claim.

As described in Section 2, scheduling may impact OLTP performance because the UNIX scheduler has no knowledge of what is happening inside of transactions. It is not clear what is the best or at least good behavior of UNIX scheduler for OLTP. The question gets more complicated in a

shared memory multiprocessor system (MP). [6] reports that a performance bottleneck has been eliminated by affinity scheduling on Sequent's MP system. This is because caches in MP machines play a much more important role than in a uniprocessor system, and the cache utilization is heavily influenced by the scheduler. We are investigating the scheduler issues with respect to OLTP performance, including affinity.

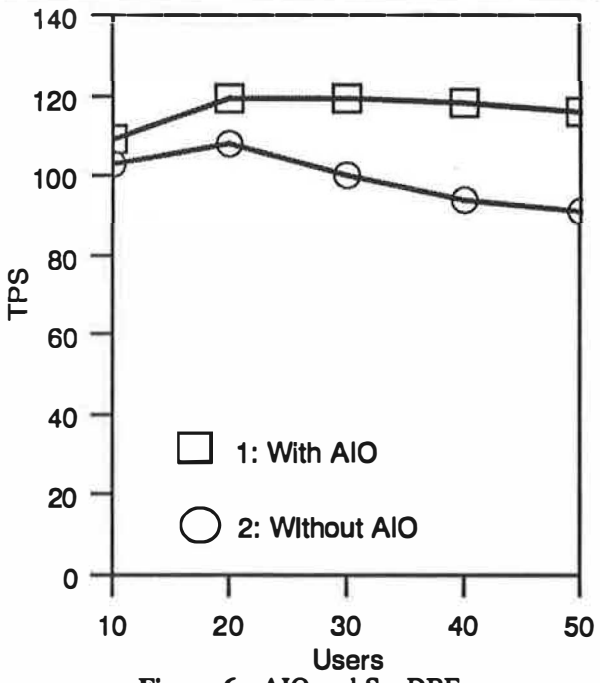


Figure 6: AIO and SunDBE

Conclusion

OLTP performance is determined by many factors. Among them, we examined two areas that are critical to OLTP performance: virtual memory and the I/O system. The shared page table feature has been implemented to enhance the SunOS VM system, and shows that it can effectively support a large number of user processes with minimal performance degradation. We also implemented fast asynchronous I/O for a high performance I/O system, and verified through extensive experiments that asynchronous I/O outperforms synchronous I/O for an OLTP workload. The improvement increases as the workload gets heavier. We have shown that the shared page table and asynchronous I/O features provide an effective way of improving OLTP performance on UNIX.

Acknowledgement

We would like to acknowledge Vlada Matena as one of key implementors. Special thanks to Steve Kleiman, Jim Skeen, and David Rosenthal who read this paper and provided many useful comments. Gyan Bhal helped with drawing the figures.

References

[1] R. Gingell, J. Moran, and W. Shannon, "Virtual Memory Architecture in SunOS", *Proceedings of the USENIX 1987 Summer Conference*, Phoenix, 1987.

[2] AT&T, *System V Interface Definition*, Volume I, 1986.

[3] J. Moran, "SunOS Virtual Memory Implementation", *Proceedings of the Spring 1988 EUUG Conference*, London, England, 1988.

[4] H. Chartock and P. Snyder, "Virtual Swap Space in SunOS", *Proceedings of Autumn 1991 EUUG Conference*, Budapest, September, 1991.

[5] M. Stonebraker, "Operating System Support for Database Management", *Communications of ACM*, July 1981.

[6] S. Thakkar and M. Swseiger, "Performance of an OLTP Application on Symmetry Multiprocessor System", *Proceedings of International Symposium on Computer Architecture*, May 1990.

[7] Transaction Processing Performance Council, "TPC Benchmark B", Standard Specification, Rev 1.1, Shanley Public Relations, San Jose, CA, March 1, 1990.

[8] M. Seltzer and M. Olson, "LIBTP: Portable, Modular Transactions for UNIX", *Proceedings of the USENIX 1992 Winter Conference*, San Francisco, 1992.

[9] E. Cheng, *et. al*, "An Open and Extensible Event-based Transaction Manager", *Proceedings of the USENIX 1991 Summer Conference*, Nashville, 1991.

[10] M. Young, *et. al*, "A Modular Architecture for Distributed Transaction Processing", *Proceedings of the USENIX 1991 Winter Conference*, Dallas, 1991.

[11] Pyramid Technology Corporation, *OSx 4BSD Manual Pages, Volume II*, May 1990.

[12] A. Buck and R. Coyne Jr., "An Experimental Implementation of Draft POSIX Asynchronous I/O", *Proceedings of the USENIX 1991 Winter Conference*, Dallas, 1991.

[13] Sun Microsystems, *Sun Database Excelerator 1.3 Release Manual*, 1992.

[14] S. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX", *Proceedings of the USENIX 1986 Summer Conference*, Atlanta, 1986.

[15] D. Rosenthal, "Evolving the Vnode Interface", *Proceedings of the USENIX 1990 Summer Conference*, Anaheim, 1990.

[16] SPARC Architecture Manual Version 8, SPARC International, Menlo Park CA, 1991.

[17] J. Gray (ed.), *The Benchmark Handbook for Database and Transaction Processing Systems*, Morgan Kaufman Publishers, 1991.

[18] J. McGrory II, *et. al*, "Transaction Processing Performance on PA-RISC Commercial Unix Systems", *IEEE COMPCON Digest of Papers* , February, 1992.

Author Information

Hyuck Yoo is a Member of Technical Staff at Sun. He received a Ph.D. in Computer Science from University of Michigan, Ann Arbor in 1990 and a B.S. in Electronic Engineering from Seoul National University, Korea in 1982. His e-mail address is hxy@Eng.Sun.COM.

Tom Rogers is the manager of the Database Engineering group at Sun. He has been with Sun for 8 years. He holds M.S. in Computer Science from Stanford University.

Reach either author via U. S. mail at 2550 Garcia Avenue, Mountain View, CA 94043.

Measurement, Analysis, and Improvement of UDP/IP Throughput for the DECstation 5000

Jonathan Kay & Joseph Pasquale[1] – University of California, San Diego

ABSTRACT

Networking software is a growing bottleneck in modern workstations, particularly for high throughput applications such as networked digital video. We measure various components of the UDP/IP protocol stack in a DECstation 5000/200 running Ultrix 4.2a, and find that checksumming and copying dominate the processing time for high throughput applications. This paper describes network software measurements and performance improvements which derive from a faster checksum implementation.

Introduction

Network software of most modern workstation operating systems is not able to take full advantage of hardware speeds. With emerging high-bandwidth network hardware and network-I/O intensive applications, operating system network software is increasingly a bottleneck. The highest-bandwidth network widely supported by workstation vendors is FDDI, which operates at 100 megabits per second or 12.5 megabytes per second (MB/s). Measuring the performance of DEC's FDDI controller for a DECstation 5000/200 at the level of the device-driver, we find that the send rate is 7-8 MB/s (the receive rate is even higher) using maximum size FDDI packets. However, when sending from user level using UDP/IP over an FDDI network, the maximum sustained throughput measures at only 2.4 MB/s. The network software is reducing throughput from user processes to the device by a factor of three.

The goal of this study is to determine how various components of network software contribute to this bottleneck. We instrument the UDP/IP protocol stack since most of the traffic on our departmental networks is comprised of NFS-generated UDP packets. We analyze network software component processing time by layer and by operation. By layer, we measure the individual processing times of the socket, UDP, IP, link, and device driver software. By operation, we measure the processing times for various copying, checksumming, and malloc/free operations. Other studies have shown these operations to be expensive [3-5, 7]. Cabrera et al. discuss a number of network software bottlenecks in the 4.2 release of Berkeley Unix [3]. Clark notes that checksumming and to a lesser extent copying were the dominant costs in the Multics TCP/IP implementation [4]. Clark et al. discuss copy and checksum costs versus other protocol costs in the context of a precursor to the 4.3 Reno release of Berkeley Unix and an experiment that attempted to isolate protocol processing costs [5]. Jacobson discusses a number of costs in the 4.3 Tahoe release of Berkeley Unix and their fixes in the Reno precursor [7].

This work is part of Project Sequoia 2000 [9], a project which brings together computer scientists and global change scientists from all over the University of California, including from UC San Diego and UC Berkeley, to improve network, storage, database, and visualization support for global change research. An important goal of the Sequoia 2000 network [6] which we are constructing is to support high throughput for the large volumes of data required for global change applications. The research described here furthers that goal by improving the effective networking bandwidth available to Sequoia 2000 researchers.

Experiments

We connect an HP 1652B Logic Analyzer to the DECstation TURBOchannel to obtain software processing time measurements with a resolution of 40 nanoseconds (which is the DECstation clock cycle time). Special cpp macros are placed at the beginning and end of the source code of each operation of interest. Each time a macro executes, a pattern plus an event identifier is sent over the DECstation's I/O bus. The logic analyzer is programmed to recognize the pattern and store the event identifier, along with a timestamp. The measurement software causes minimal interference, generating overhead of less than 1% of the total network software processing time. Statistical analysis is done off line.

The experimental system consists of two workstations connected by an FDDI network with no other workstations and no network traffic other than that generated by the experiment. An experiment consists of one workstation sending a message to the system under test , which then sends the same message back. All measurements are made on the system under test, which is executing a probed kernel

[1]This work was supported in part by grants from DEC, IBM, NCR, NSF, TRW, and UC MICRO.

and is hooked up to the logic analyzer. The experiments are repeated 100 times for each message size to obtain statistical significance in the results. That experiment, in turn, is repeated for 40 message sizes in the range between 1 and 8192 bytes. It is worth noting that NFS typically transfers large files in units of 8192 byte blocks.

Measurements

We present the total message processing time for various components of network software when receiving and then sending the same-sized message, by layer and by operation. For example, a checksum time of 359 microseconds for a 1024 byte message refers to the total time spent in the six calls to the checksum routine: two header and one data checksum computed while receiving a 1024 byte message and for sending the same message back. Since the maximum FDDI frame size is 4352 bytes, IP must break larger messages into two FDDI frames on the sending side and reassemble them on the receiving end (fragmentation/defragmentation).

Analysis by Operation

We analyze operation overheads in two different ways. The first is to look at the overall operation times over both send and receive sides. This is relevant to total processing overhead and latency. However, since a major goal of this paper is to understand bottlenecks, and throughput is limited by max(send time, receive time), we also analyze operation times by send side vs. receive side.

Total Operation Costs

Figure 1 shows the per-message combined send and receive processing times for various operations: CHECKSUM, COPY, MALLOC/FREE, and OTHER (i.e., the combined times for all other operations). Figure 2 shows cumulative times for each operation (so that the top line is total processing time for messages of given sizes). Figure 3 is another cumulative graph of components of COPY.

CHECKSUM refers to computation of the Internet checksum in UDP and IP. Processing time for CHECKSUM increases with message byte size at a rate of approximately 0.330 μsec per byte. For large messages, CHECKSUM processing time dominates that of all other operations; as can be seen in Figure 2, it is responsible for almost 50% of the total processing time for 8192 byte messages.

COPY refers to various bulk data copying operations. Figure 3 contains a breakdown of the various components of COPY: copying between user and kernel memories (35%), copying from the kernel to the FDDI controller across the relatively slow I/O bus (50%), cache clearing in preparation for DMA from the FDDI controller to kernel memory (10%). The values in parentheses refer to the percentage of total COPY time for which the operation is responsible for all but the smallest message sizes. Total processing time for COPY increases with message byte

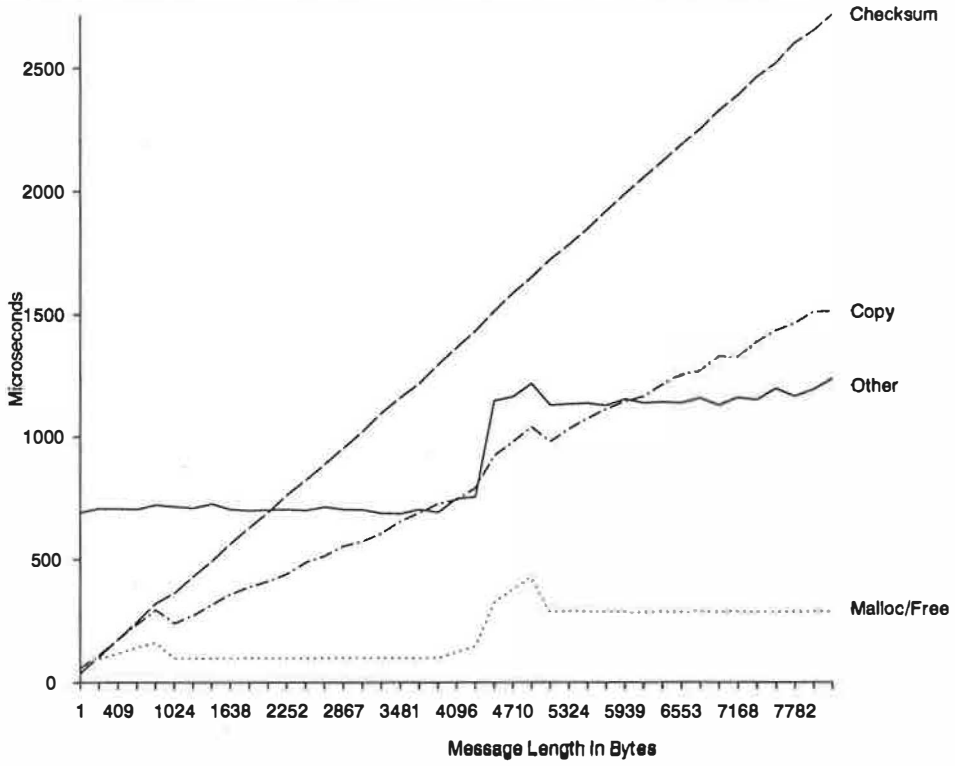


Figure 1: Aggregate operation times

size at a rate of approximately 0.175 μsec per byte, or about half the rate of CHECKSUM.

MALLOC/FREE refers to allocation and freeing of memory buffers. Berkeley UNIX, of which Ultrix is a derivative, stores messages in data structures called mbufs. An mbuf comprises a header, which can directly store up to 108 bytes of data, and possibly a page (4096 bytes) pointed to by the header, which can be used to store data of size greater than 108 bytes; if the data page is used, no data is stored in the header. To reduce internal fragmentation, the socket sending code allocates a string of up to 10 small mbufs if the message size is less than 1024 bytes. For messages larger than 1024 bytes but less than a 4096 byte page, an mbuf with one data page is allocated. A number of operations, including both malloc and free, must be performed on each mbuf. Consequently, the processing time for MALLOC/FREE increases with the number of 108 byte size blocks up to 1024 bytes, and then is

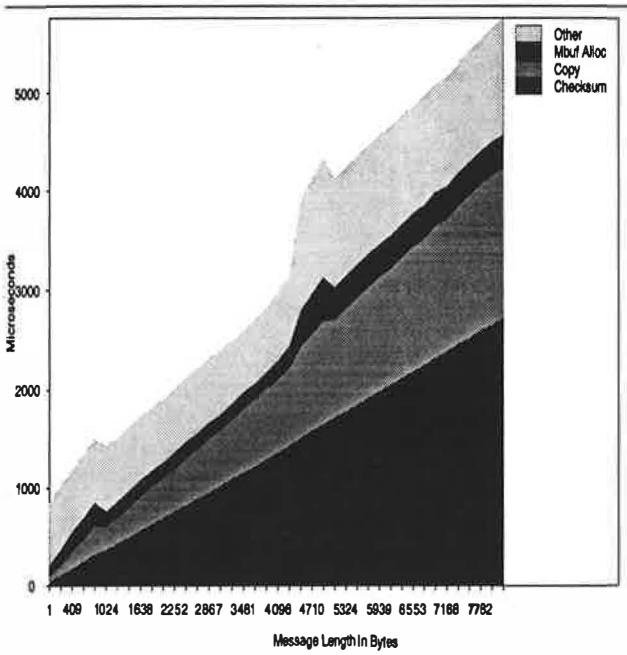


Figure 2: Accumulated Operation Times

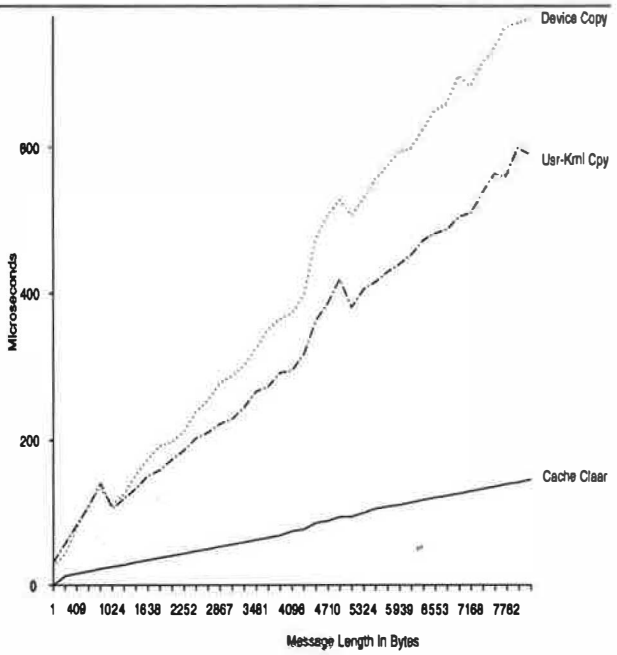


Figure 3: Copy Times

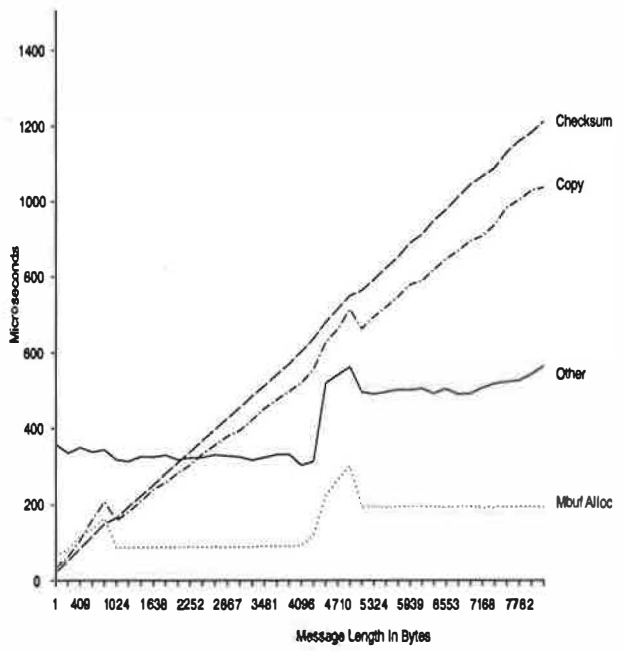


Figure 4: Send side operation times

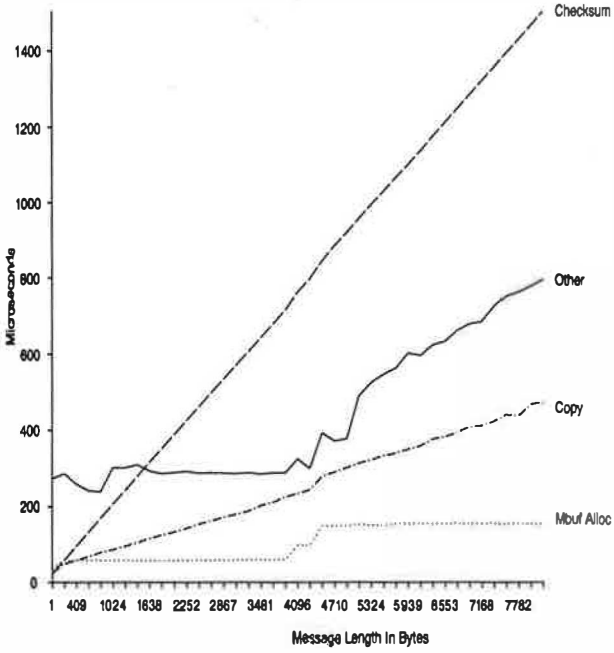


Figure 5: Receive side operation times

constant for messages sizes which fit in one page. In a similar fashion, for messages larger than a page, another string of mbufs are allocated for the data between 4096 bytes and 5120 bytes, whereas for messages larger than 5120 bytes, two data pages are allocated. That is the reason for the humps, both in MALLOC/FREE and elsewhere, for sizes just less than 1024 bytes and 5120 bytes, and the constant times for messages with sizes between 1024 and 4096 bytes and between 5120 and 8192 bytes. Relative to total processing time, MALLOC/FREE accounts for 3-7%, but can go higher than 10% at the humps. However, this time is the aggregate of many mbuf allocation times; individual malloc operations are very cheap. Upon inspection of the Ultrix source code, it is obvious that a great deal of effort went into optimizing the speed of malloc and free.

OTHER refers to all other processing not accounted for by CHECKSUM, COPY, and MALLOC/FREE. This includes protocol processing, various mbuf operations (e.g., header prepending, queueing/dequeueing), fragmentation/defragmentation, software interrupts, checking system call parameters, etc. As can be seen from Figures 1 and 2, OTHER appears to increase with the number of mbufs in a similar fashion as MALLOC/FREE, except that it accounts for a much greater percentage of the total processing time than MALLOC/FREE.

Costs of Sending vs. Receiving

This subsection analyzes the breakdown of operation times into times spent to send messages vs. times spent to receive messages. Figures 4 and 5 show the per-message send and receive processing times, respectively. Figures 6 and 7 show the send and receive cumulative times. Figures 8 and 9 contain graphs of components of COPY. One can see from Figures 6 and 7 that the send-side and receive-side times are fairly well balanced, except at the humps caused by the mbuf allocation algorithms, where the receive side is noticeably faster than the send side.

CHECKSUM is the dominant operation for large messages on both send and receive sides. The receive side costs rise more steeply with message size (0.180 μsec/byte) than so the send side costs (0.150 μsec/byte). This is because messages being sent are already in the cache, whereas incoming messages are put into memory (and not in the cache) by DMA, and are first touched by the checksum code. Thus, the checksumming time shown includes the time of moving data from memory into the cache.

COPY time is much larger on the sending side than on the receiving side. Figures 8 and 9 contain breakdowns of COPY times on each side. Time spent copying between user and kernel space is approximately the same for each side, because in each case the message is already cached. However, on the sending side the dominant time is due to copying data to the FDDI interface, since the FDDI interface device does not support DMA on the send side. On the other hand, the receive side must perform a cache clear in preparation for incoming data transferred by DMA (which is supported by the interface on the receive side). Send-side and receive-side times remain balanced largely because

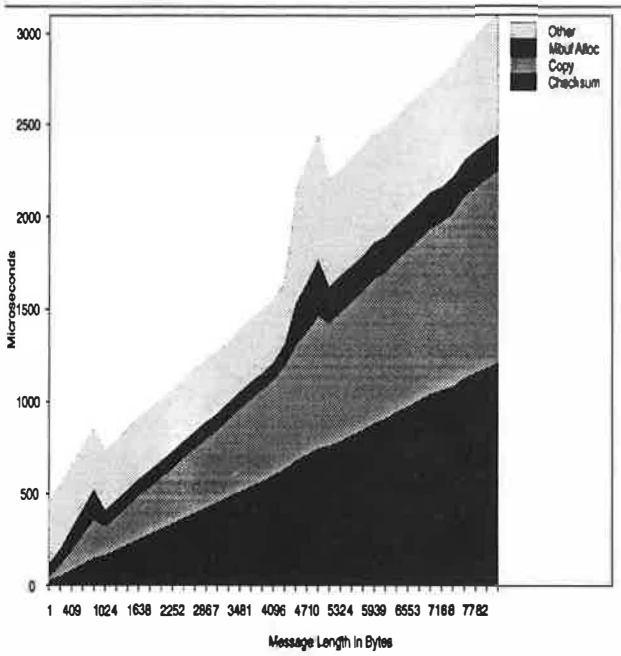


Figure 6: Accumulated send side operation times

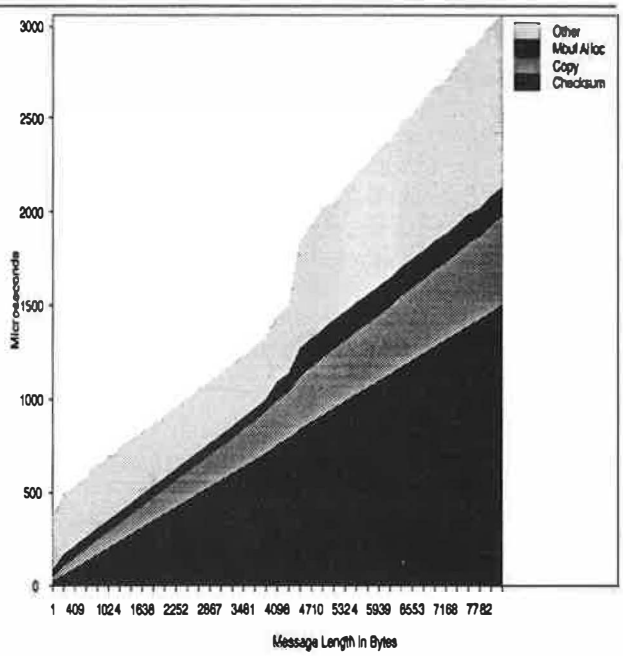


Figure 7: Accumulated receive side operation times

the time to copy to the device is offset by the receive-side checksumming time and the receive-side cache clear operation time.

The effects of the complicated mbuf MALLOC/FREE described above are easy to see here. Only the send side features the humps caused by the send-side mbuf allocation algorithm. On the receive side, the shape of the MALLOC/FREE curve is caused by the FDDI driver allocation policy. The FDDI driver allocates a single small mbuf for incoming frames less than 108 bytes in length, a cluster mbuf for frames between 108 and 4096 bytes, a cluster mbuf and a small mbuf for incoming frames between 4096 and 4204 bytes in length, and two cluster mbufs for larger frames. Since messages larger than the FDDI MTU size of 4324 bytes are fragmented into two FDDI frames, the algorithm repeats with additional mbufs for messages larger than 4324 bytes.

As in the combined case analyzed above, OTHER largely shows the same profile as MALLOC/FREE, but on a larger scale. When the receive-side times are isolated from the send-side times, it becomes possible to see that OTHER is climbing with message size above the FDDI MTU. Since we know little about OTHER, we cannot account for that phenomenon.

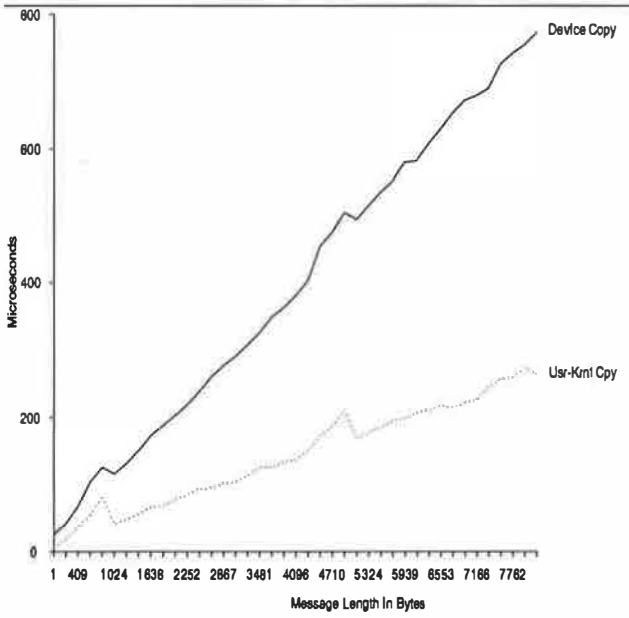


Figure 8: Send side copy times

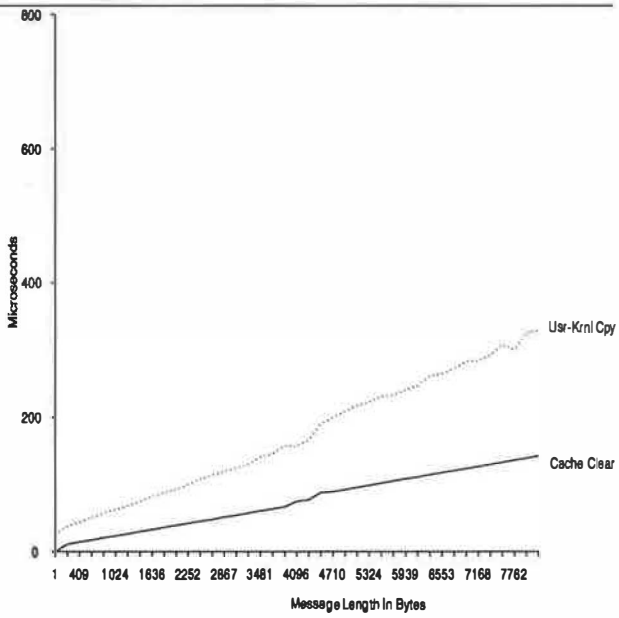


Figure 9: Receive side copy times

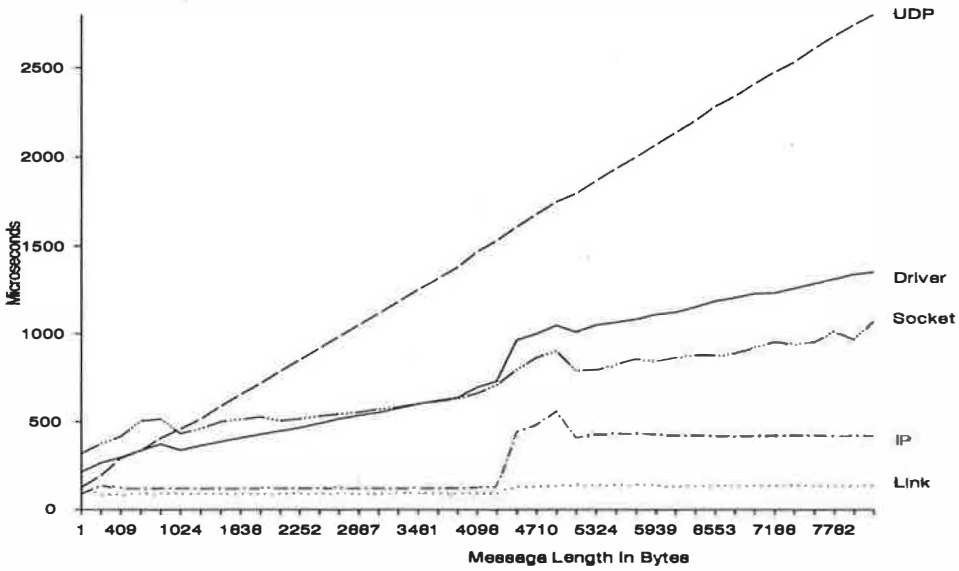


Figure 10: Layer times

Analysis by Layer

Figure 10 shows the per-message combined send and receive processing times for various layers: SOCKET, UDP, IP, LINK, and DRIVER.

It is obvious that UDP is the most time-consuming layer. This is mainly due to its checksum computation; UDP is the only layer that computes a checksum for the data portion of its messages. The next most time-consuming layer is the DRIVER. This is largely due to its copy operation from the kernel to the controller, which is the most significant of all the copy operations. The SOCKET layer processing time is due to copy operations between user and kernel memories, as well as OTHER operations. By contrast, the IP layer does not contain significant checksum or copy operations, and therefore its processing time does not increase linearly with message size like the previous layers. However, since fragmentation/defragmentation occur in the IP layer, we see a significant jump in processing time when the messages requires more than one packet. Finally, the LINK layer refers to code lying between the IP and DRIVER layers. This includes data-link encapsulation code common to all Ethernet and FDDI devices, and software interrupt code. Like the IP layer, its processing time increases with number of fragments per message rather than message size. This is because the number of times it is called corresponds to the number of FDDI frames which must be sent or received. However, this time is relatively small compared to that of other layers.

Checksum Improvement

A goal of this study is to improve throughput. Since CHECKSUM dominates the processing time for large messages, such as those produced by network-I/O intensive applications, this is clearly an operation worth improving. Thus, we concentrated on reducing, and where it is reasonable to do so, eliminating, checksum computation time. The first step was to optimize the checksum code itself; we found that the checksum code used in DECstations and indeed most MIPS processor-based workstations was poorly tuned for RISC processors. Below we describe our improved checksum algorithm for RISC processors, which cuts the checksum computation time in half. Since checksumming still consumed a large portion of processing time, we developed a proposal for eliminating checksumming entirely over most messages.

A RISC Checksum Algorithm

The Internet Request for Comments describing the Internet checksum [2] includes a generic checksum algorithm written in 'C' and optimized algorithms written in assembly language for three different machine architectures: a CISC microprocessor (the Motorola 68020), a vector supercomputer (the Cray), and a CISC mainframe (the IBM 3090). None of the algorithms described are well suited to RISC processors. The 68020-based algorithm relies on extended integer arithmetic support not usually found in RISC processors: a carry bit and an instruction that adds the carry bit to two operands are required. The Cray algorithm uses vector operations, an even less common feature of RISC microprocessors. The IBM algorithm uses a number of branches which causes inefficient operation for pipelined processors such as RISC microprocessors. It is not surprising that the checksum routine provided by MIPS is effectively a hand translation of the 'C' algorithm into MIPS assembly language.

We were able to make a number of improvements to that algorithm. The most effective improvement resulted from reading memory in units of 32 bit words rather than in the units of 16 bit words used by the 'C' algorithm described in [2]. Reading a 32 bit word results in a single memory access, but requires two additional instructions (shift and mask) to unpack the desired two 16 bit quantities. As long as the ratio of memory access time to instruction execution time is more than 2 (generally the case for RISC workstations, and this ratio is expected to increase), a single memory access and two register instructions are executed faster than two direct 16 bit word memory accesses.

The next most useful technique is loop unrolling, used by the Motorola and IBM algorithms [2]. The inner loop of our checksum algorithm is unrolled sixteen times. We chose sixteen because it is the maximum number of unrolls that is both a power of two, and allows the expanded loop to efficiently operate on a 108-byte small mbuf. Both the Motorola and IBM algorithms unrolled their inner loops sixteen times as well.

Some smaller improvements are derived from consideration of pipelining effects. The MIPS processor uses load delay slots to permit the processor to accomplish useful work while bringing in memory contents to a register. Unfortunately, the assembler does a poor job of scheduling the instructions of the checksum computation into those delay slots, so we resorted to hand placement. The key principle is that a register load should begin right after its previous contents are used, and therefore, multiple registers are used to achieve pipelining [8]. We use two registers in our data pipeline which is enough to keep the processor busy. This technique is independent of whatever mechanism is used to wait for memory contents to arrive at the processor (e.g., load delay slots or scoreboarding).

With these modifications, the checksum computation time was reduced by more than a factor of two. Consequently, when sending large messages, we observed a 33% overall improvement in UDP/IP throughput.

Figure 11 shows the code of the main loop of the improved checksum routine. For brevity, we leave out details such as setup, termination, and code for dealing with small or unaligned memory segments.

Checksum Elimination

Although we have made considerable gains by carefully optimizing the checksum algorithm, eliminating the checksum altogether would be twice as effective. In fact, options already exist in Berkeley Unix based networking code to completely turn off checksumming whenever interoperability would not be affected. However, the Internet checksum exists for a good reason; very simply, packets are occasionally corrupted during transmission, and the checksum is needed to detect corrupted data. Turning off checksumming by default is specifically forbidden within the Internet [1].

Eliminating Checksum Redundancy

There is a certain amount of redundant checksumming in the system. Ethernet and FDDI networks implement their own 16-bit cyclic redundancy checksum. Thus, packets sent directly over an Ethernet or FDDI network are already protected from data corruption. It is obvious from this study that this redundancy is expensive. Thus, we suggest that in cases of complete redundancy, the Internet checksum not be performed. One must be careful, though, about deciding when the Internet checksum is in fact redundant. We believe that it is not unreasonable to turn off checksums when crossing a single network which implements its own checksum. Since the destinations of most TCP and UDP messages are within the same LAN on which they are sent, this policy would eliminate checksumming on most TCP and UDP messages.

Such a policy differs somewhat from traditional TCP design in one aspect of protection against corruption. Always performing a checksum in software in host memory protects against errors in data transfer over the I/O bus in addition to the protection between network interfaces given by the Ethernet and FDDI checksums. However, since data transfers over the I/O bus for such common devices such as disks are routinely assumed to be correct and are not checked in software, we conclude that such a reduction in protection against I/O bus transfer errors for network devices is not unreasonable, especially given the heavy cost of checksum computation.

```
    lw  t5,0(a0)          # load up pipeline
    lw  t6,4(a0)          # with 32-bit words
                          # note: t5 and t6 alternate
1:                        # do {
    srl     v1,t5,16      # get upper half of 32-bit word
    addu    v0,v1         # add to running checksum
    addu    v1,t5,0xffff  # get lower half of 32-bit word
    addu    v0,v1         # add to running checksum
    lw      t5,8(a0)      # immediately start reloading t5
    srl     v1,t6,16      # get upper half of 32-bit word
    addu    v0,v1         # add to running checksum
    addu    v1,t6,0xffff  # get lower half of 32-bit word
    addu    v0,v1         # add to running checksum
    lw      t6,12(a0)     # immediately start reloading t6
    srl     v1,t5,16      # get upper half of 32-bit word
    addu    v0,v1         # add to running checksum
    addu    v1,t5,0xffff  # get lower half of 32-bit word
    addu    v0,v1         # add to running checksum
    lw      t5,16(a0)     # immediately start reloading t5
  (... 12 repetitions...)
    srl     v1,t6,16      # get upper half of 32-bit word
    addu    v0,v1         # add to running checksum
    addu    v1,t6,0xffff  # get lower half of 32-bit word
    addu    v0,v1         # add to running checksum
    lw      t6,72(a0)     # immediately start reloading t6
    addu    a0,64
    subu    a1,64
    bge     a1,72,1b      # } while (len >= 72)
```

Figure 11: Improved checksum code

However, turning off checksum protection in any wider area context seems unwise without considerable study. Not all networks are protected by checksums, and it is difficult to see how one might check that an entire routed path is protected by checksums without undue complications involving IP extensions. A more fundamental problem is that network checksums only protect a frame between network interfaces; errors may arise while a frame is in a gateway machine. Although such corruption is unlikely for a single machine, the chance of a packet not being corrupted decreases exponentially with the number of gateways a packet travels through.

In addition to disabling checksumming on local packets, we believe that it should be possible for applications to disable checksumming on any TCP connection on a per-connection basis. While disabling checksums is unwise for many applications such as NFS or FTP, many interactive audio and video applications may be tolerant of some corruptions because of the transient usage of the data.

UDP Implementation

The basic implementation in UDP is to use the existing checksum-nonuse protocol. In UDP, if the checksum field on an incoming message is zero, then it it is assumed that there is no checksum on the message. All that remains, then, is to decide on the details of when the sender should send a zeroed checksum field. Our implementation involves adding a flag to each network interface which supports some form of checksum. Under Berkeley Unix, routing information is already available to UDP, and the route structure includes a flag on whether the next hop in the route is just a gateway or is the destination host. The algorithm checks whether the next hop is a destination, and whether the network interface hardware supports checksumming. Checksumming is skipped if both of those conditions are true. We also implemented a new socket option which lets the application turn off checksumming, even for sessions involving a large number of hops between hosts, useful for audio and video applications.

TCP Implementation

Extending TCP to implement checksum elimination is more difficult, since one is always supposed to checksum TCP data, and zero is specifically a valid checksum. Thus, some additional mechanism is required so that two implementations can agree on turning off checksums. An experimental Alternate Checksum Option already exists [11]. The Alternate Checksum Option provides a generic mechanism for TCP implementations to agree on use of an alternate checksum, as well as mechanism in case an alternate checksum requires more space than is provided by the TCP checksum field (e.g., a 32-bit or 64-bit checksum). Thus, it is straightforward to define an alternate checksum value specifying no data checksum.

Negotiation is started by the side that wishes to create a connection. TCP uses the same series of checks as in UDP to decide whether to send an option to turn off checksums. When server-side TCP responds to a TCP connection request, it will respond with the option if it gets a checksum-elimination request in the connection setup packet. Each side of a conversation only disables checksumming if it gets a checksum elimination option sent by the other host in the conversation.

As with UDP, we added a socket option that causes the implementation to try to negotiate turning off checksums. Notice that since the sender of the option makes the decision as to whether to negotiate checksums, and since receivers knowledgeable of this option automatically agree to any suggestion of turning off checksums, only one side of a conversation need call the option.

Performance

Table 1 summarizes the throughputs resulting from the improved checksum techniques. The numbers in MB (i.e., 1,000,000 bytes) are absolute throughputs of each configuration, and the percentage values in parenthesis are the percentage improvements relative to unmodified Ultrix 4.2a. The performance improvements are significant.

A Faster FDDI Adapter

Earlier we noted that the FDDI adapter we are using to perform our experiments does not support outgoing DMA, therefore, the processor must copy outgoing frames to the controller. Furthermore, we showed in Figure 8 that copying data out to the device (the DEVICE COPY operation) is time-consuming for large messages. DEC has recently developed a new FDDI controller for the DECstation which features send-side DMA, which we obtained after conducting the experiments already described. In running some tests using the new controller, we learned, unsurprisingly, that send-side performance using the new controller is much better than with the old controller. The difference is even more marked

	TCP	UDP
Unmodified Ultrix 4.2a	2.1 MB/s	2.4 MB/s
Optimized Checksum	2.7 MB/s (29%)	3.2 MB/s (33%)
Checksum Elimination	3.1 MB/s (48%)	4.1 MB/s (71%)

Table 1: Throughput Improvements From Checksum Optimization and Elimination

once one eliminates the overhead of checksumming. Table 2 lists send-side UDP/IP throughputs using the old and new controllers, without and with checksum elimination. Note that the throughputs given in Table 2 using the old adapter and unmodified kernel are higher than those in Table 1. This is because Table 2 describes only send-side performance, while the experimental configuration used to generate Table 1 involved the receive side which is often the bottleneck.

Notice that our checksum elimination technique results in a 58% improvement in throughput (4.1 vs. 2.6 MB/s) using the old adapter, and a 91% improvment (6.3 vs. 3.3 MB/s) using the new adapter. This is because the new adapter increases the speed of data transfer from memory to the network device (due to DMA, which can operate concurrently with other network processing), making the checksum computation time, and therefore its elimination, a more prominent factor.

Conclusions

We present measurements of DECstation UDP/IP network software, analyzed by operation and by layer. The main time consuming operations, particularly for large messages, are the checksum computation and, to a lesser degree, data copying. The main time-consuming layer is the UDP transport layer, since it contains the checksum computation. We improved the checksum implementation, and suggested a way of eliminating checksum computations which could be considered redundant. Both of these improvements resulted in a noticeable increase in throughput performance.

Acknowledgements

The section on improved adapter performance was made possible by Fred Templin's tenacity in getting us the new FDDI adapter and a driver for it. Keith Muller was of great assistance in setting up the logic analyzer and designing probe software. Kevin Fall aided in bulletproofing the checksum elimination algorithm.

Availability

Improved checksum software is available in the form of diffs against Ultrix 4.2a. It can be found on the ftp server ucsd.edu.

References

[1] R. T. Braden, "Requirements for Internet Hosts – Communication Layers," Internet Request for Comments 1122, October 1989.

[2] R. T. Braden, D. A. Borman, and C. Partridge, "Computing the Internet Checksum," Internet Request for Comments 1071, September 1988.

[3] L.-F. Cabrera, E. Hunter, M. J. Karels, D. A. Mosher, "User-Process Communication Performance in Networks of Computers," IEEE Transactions on Software Engineering, 14(1), 38-53, Jan. 1988.

[4] D. D. Clark, "Modularity and Efficiency in Protocol Implementation", Internet RFC, 817, 1982

[5] D. D. Clark, V. Jacobson, J. Romkey, H. Salwen, "An Analysis of TCP Processing Overhead," IEEE Communications Magazine, 23-29, June 1989.

[6] D. Ferrari, J. Pasquale, and G. Polyzos, "Network Issues for Sequoia 2000," Proceedings IEEE COMPCON, 401-406, February 1992.

[7] V. Jacobson, "BSD TCP Ethernet Throughput", comp.protocols.tcp-ip, Usenet, 1988.

[8] J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[9] M. Stonebraker and J. Dozier, "Sequoia 2000: Large Capacity Object Servers to Support Global Change Research," Sequoia 2000 Technical Report #1, U.C. Berkeley, July 1991.

[10] R. W. Watson, S. A. Mamrak, "Gaining Efficiency in Transport Services by Appropriate Design and Implementation Choices," ACM Transactions on Computer Systems, 5(2), 97-120, May 1987.

[11] J. Zweig and C. Partridge, "TCP Alternate Checksum Options," Internet Request for Comments 1146, March 1990.

Author Information

Jonathan Kay is a Ph.D. student in computer science at the University of California, San Diego. He received both B.S. and M.S. in Computer Science from the Johns Hopkins University in 1989 and 1990, respectively, and hopes to get his Ph.D from UCSD sometime this decade. As you might expect from this paper, he is interested in TCP/IP performance improvements, gigabit networking, network support for multimedia, and data movement issues in general. He works in the UCSD Computer Systems

	Old Adapter	New Adapter
Unmodified Ultrix 4.2a	2.6 MB/s	3.3 MB/s
Checksum Elimination	4.1 MB/s	6.3 MB/s

Table 2: Send-Side throughput improvements from The improved adapter

Lab, and is currently involved in making the Sequoia 2000 network faster. His e-mail address is jkay@cs.ucsd.edu.

Joseph Pasquale is an Assistant Professor of Computer Science and Engineering at the University of California, San Diego, where he co-directs the UCSD Computer Systems Laboratory. He is also a Senior Fellow at the San Diego Supercomputer Center. Dr. Pasquale received the B.S. and M.S. degrees in Computer Science from MIT in 1982, and the Ph.D. degree in Computer Science from U.C. Berkeley in 1988. He is a recipient of the NSF Presidential Young Investigator Award in 1989 and the IBM Faculty Development Award in 1991. His current research interests are in the areas of operating systems and networks, particularly the design, implementation, and performance evaluation of I/O system and network software to support distributed multimedia (especially digital video and audio) applications and I/O-intensive scientific applications. Dr. Pasquale is currently involved in the design of the Sequoia 2000 Network, which connects the University of California campuses, and whose goals are to support high throughput transmission of scientific data and to provide real-time support for collaborative distributed multimedia applications such as video conferencing. Reach him electronically at pasquale@cs.ucsd.edu.

Both authors can be reached via U. S. mail at Computer Systems Laboratory; Department of Computer Science and Engineering; University of California, San Diego, La Jolla, CA 92093-0114.

The BSD Packet Filter: A New Architecture for User-level Packet Capture

Steven McCanne & Van Jacobson – Lawrence Berkeley Laboratory[1]

ABSTRACT

Many versions of Unix provide facilities for user-level packet capture, making possible the use of general purpose workstations for network monitoring. Because network monitors run as user-level processes, packets must be copied across the kernel/user-space protection boundary. This copying can be minimized by deploying a kernel agent called a *packet filter*, which discards unwanted packets as early as possible. The original Unix packet filter was designed around a stack-based filter evaluator that performs sub-optimally on current RISC CPUs. The BSD Packet Filter (BPF) uses a new, register-based filter evaluator that is up to 20 times faster than the original design. BPF also uses a straightforward buffering strategy that makes its overall performance up to 100 times faster than Sun's NIT running on the same hardware.

Introduction

Unix has become synonymous with high quality networking and today's Unix users depend on having reliable, responsive network access. Unfortunately, this dependence means that network trouble can make it impossible to get useful work done and increasingly users and system administrators find that a large part of their time is spent isolating and fixing network problems. Problem solving requires appropriate diagnostic and analysis tools and, ideally, these tools should be available where the problems are – on Unix workstations. To allow such tools to be constructed, a kernel must contain some facility that gives user-level programs access to raw, unprocessed network traffic [7]. Most of today's workstation operating systems contain such a facility, e.g., NIT[10] in SunOS, the Ultrix Packet Filter [2] in DEC's Ultrix and Snoop in SGI's IRIX.

These kernel facilities derive from pioneering work done at CMU and Stanford to adapt the Xerox Alto 'packet filter' to a Unix kernel[8]. When completed in 1980, the CMU/Stanford Packet Filter, CSPF, provided a much needed and widely used facility. However on today's machines its performance, and the performance of its descendents, leave much to be desired – a design that was entirely appropriate for a 64KB PDP-11 is simply not a good match to a 16MB Sparcstation 2. This paper describes the BSD Packet Filter, BPF, a new kernel architecture for packet capture. BPF offers substantial performance improvement over existing packet capture facilities – 10 to 150 times faster than Sun's NIT and 1.5 to 20 times faster than CSPF on the same hardware and traffic mix. The performance increase is the result of two architectural improvements:

- BPF uses a re-designed, register-based 'filter machine' that can be implemented efficiently on today's register based RISC CPU. CSPF used a memory-stack-based filter machine that worked well on the PDP-11 but is a poor match to memory-bottlenecked modern CPUs.
- BPF uses a simple, non-shared buffer model made possible by today's larger address spaces. The model is very efficient for the 'usual cases' of packet capture.[2]

In this paper, we present the design of BPF, outline how it interfaces with the rest of the system, and describe the new approach to the filtering mechanism. Finally, we present performance measurements of BPF, NIT, and CSPF which show why BPF performs better than the other approaches.

The Network Tap

BPF has two main components: the network tap and the packet filter. The network tap collects copies of packets from the network device drivers and delivers them to listening applications. The filter decides if a packet should be accepted and, if so, how much of it to copy to the listening application.

Figure 1 illustrates BPF's interface with the rest of the system. When a packet arrives at a network interface the link level device driver normally sends it up the system protocol stack. But when BPF is listening on this interface, the driver first

[1]This work was supported by the Director, Office of Energy Research, Scientific Computing Staff, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

[2]As opposed to, for example, the AT&T STREAMS buffer model used by NIT which has enough options to be Turing complete but appears to be a poor match to any practical problem.

calls BPF. BPF feeds the packet to each participating process' filter. This user-defined filter decides whether a packet is to be accepted and how many bytes of each packet should be saved. For each filter that accepts the packet, BPF copies the requested amount of data to the buffer associated with that filter. The device driver then regains control. If the packet was not addressed to the local host, the driver returns from the interrupt. Otherwise, normal protocol processing proceeds.

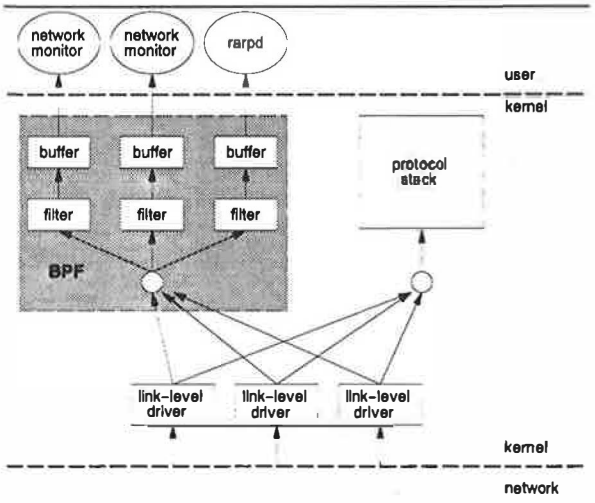


Figure 1: BPF Overview

Since a process might want to look at every packet on a network and the time between packets can be only a few microseconds, it is not possible to do a read system call per packet and BPF must collect the data from several packets and return it as a unit when the monitoring application does a read. To maintain packet boundaries, BPF encapsulates the captured data from each packet with a header that includes a time stamp, length, and offsets for data alignment.

Packet Filtering

Because network monitors often want only a small subset of network traffic, a dramatic performance gain is realized by filtering out unwanted packets in interrupt context. To minimize memory traffic, the major bottleneck in most modern workstations, the packet should be filtered 'in place' (e.g., where the network interface DMA engine put it) rather than copied to some other kernel buffer before filtering. Thus, if the packet is not accepted, only those bytes that were needed by the filtering process are referenced by the host.

In contrast, SunOS's STREAMS NIT [10] copies the packets before filtering and as a result suffers a performance degradation. The STREAMS packet filter module (*nit_pf(4M)*) sits on top of the packet capture module (*nit_if(4M)*). Each packet received is copied to an mbuf, and passed off to NIT, which then allocates a STREAMS message buffer and copies in the packet. The message buffer is then sent upstream to the packet filter, which may decide to discard the packet. Thus, a copy of each packet is always made, and many CPU cycles will be wasted copying unwanted packets.

Tap Performance Measurements

Before discussing the details of the packet filter, we present some measurements which compare the relative costs of the BPF and SunOS STREAMS buffering models. This performance is independent of the packet filtering machinery.

We configured both BPF and NIT into the same SunOS 4.1.1 kernel, and took our measurements on a Sparcstation 2. The measurements reflect the overhead incurred during the interrupt processing – i.e., how long it takes each system to stash the packet into a buffer. For BPF we simply measured the before and after times of the tap call, *bpf_tap()*, using the Sparcstation's microsecond clock. For NIT we measured the time of the tap call *snit_intr()* plus the additional overhead of copying promiscuous packets to mbufs. (Promiscuous packets are those packets which were not addressed to the local host, and are present only because the packet filter is running.) In other words, we included the performance hit that NIT takes for not filtering packets in place. To obtain accurate timings, interrupts were locked out during the instrumented code segments.

The data sets were taken as a histogram of processing time versus packet length. We plotted the mean processing per packet versus packet size, for two configurations: an "accept all" filter, and a "reject all" filter. In the first case, the STREAMS NIT buffering module (*nit_buf(4M)*) was pushed on the NIT stream with its *chunksize* parameter set to the 16K bytes. Similarly, BPF was configured to use 16K buffers. The packet filtering module which usually sits between the NIT interface and NIT buffering modules was omitted to effect "accept all" semantics. In both cases, no truncation limits were specified. This data is shown in Figure 2. Both BPF and NIT show a linear growth with captured packet size reflecting the cost of packet-to-filter buffer copies. However the different slopes of the BPF and NIT lines show that BPF does its copies at memory speed (148ns/byte) while NIT runs 45% slower (216ns/byte).[3] The y-intercept gives the

[3]This difference is due to the fact that NIT is not as careful about alignment as BPF. The network driver wants the IP header aligned on a longword boundary, but an Ethernet header is 14 bytes so the start of the packet is shortword aligned. Since NIT copies the packet to a longword aligned boundary, an inefficient, misaligned bcopy results. This oversight will be felt twice – once in this measurement, and again at the user-level, when for instance, a network monitor like *tcpdump* or *etherfind* must copy the network-layer portion of the packet to a longword aligned boundary.

fixed per-packet processing overhead: The overhead of a BPF call is about 6 μsec, while NIT is 15 times worse at 89 μsec per packet. Much of this huge disparity appears to be due to the cost of allocating and initializing a buffer under the remarkably baroque AT&T STREAM I/O system.[4]

Figure 3 shows the results for the "reject all" configuration. Here the STREAMS packet filter module was configured with a "reject all" filter and pushed directly on top of the NIT interface module (the NIT buffering module was not used). Since the filter discards all packets, the processing time should be constant, independent of the packet size. For BPF this is true – we see essentially the same fixed cost as last time (5 μsec instead of 6 since rejecting avoids a call to the BPF copy routine) and no effect due to packet size. However, as explained earlier, NIT doesn't filter packets in place but instead copies packets then runs the filter over the copies.[5] Thus the cost of running NIT increases with packet size *even when the packet is discarded by the filter*. For large packets, this gratuitous copy makes NIT almost two orders of magnitude more expensive than BPF (450 μsec vs. 5 μsec).

The major lesson here is that filtering packets early and in place pays off. While a STREAMS-like design might appear to be modular and elegant, the performance implications of module partitioning need to be considered in the design. We claim that even a STREAMS-based network tap should include the packet filtering and buffering functionality in its lowest layer. There is very little design advantage in factoring the packet filter into a separate streams module, but great performance advantage in integrating the packet filter and the tap into a single unit.

The Filter Model

Assuming one uses reasonable care in the design of the buffering model,[6] it will be the dominant cost of packets you accept while the packet filter computation will be the dominant cost of packets you reject. Most applications of a packet capture facility reject far more packets than they accept and, thus, good performance of the packet filter is critical to good overall performance.

A packet filter is simply a boolean valued function on a packet. If the value of the function is *true* the kernel copies the packet for the application; if it is *false* the packet is ignored.

[4] You might note anomalous behavior near the origin for the NIT data in both this and the following graph. STREAMS must allocate an mblk (a STREAMS buffer descriptor) for every packet. For small packets, the packet data is copied into a region of the mblk while large packets must use a more elaborate allocator involving additional dblk ('data block') allocations.

[5] The copy is required because the filter is a separate STREAM module pushed on top of the capture module and, thus, the capture module must copy data to STREAM buffers to send it up the stream to the filter module. As was the case with capture/buffer separation, the documentation notes this capture/filter separation is a feature, not a bug.

[6] E.g., *not* STREAMS.

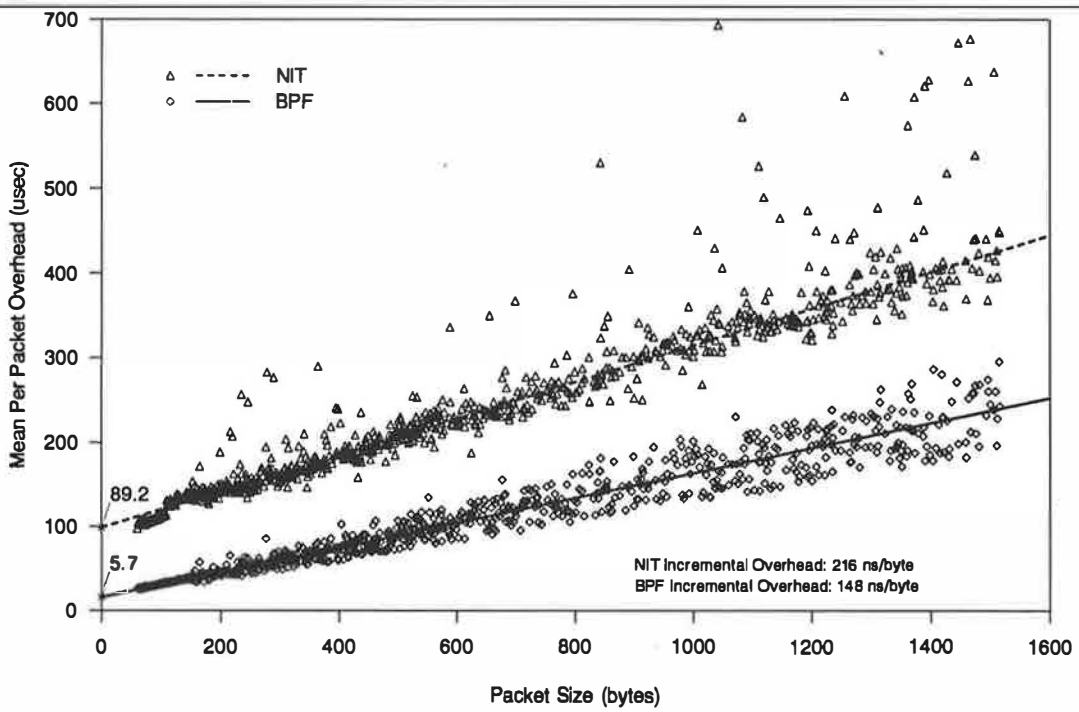


Figure 2: NIT versus BPF: "accept all"

Historically there have been two approaches to the filter abstraction: a boolean expression tree (used by CSPF) and a directed acyclic control flow graph or CFG (first used by NNStat[1] and used by BPF). For example, Figure 4 illustrates the two models with a filter that recognizes either IP or ARP packets on an Ethernet. In the tree model each node represents a boolean operation while the leaves represent test predicates on packet fields. The edges represent operator-operand relationships. In the CFG model each node represents a packet field predicate while the edges represent control transfers. The righthand branch is traversed if the predicate is true, the lefthand branch if false. There are two terminating leaves which represent *true* and *false* for the entire filter.

These two models of filtering are computationally equivalent. I.e., any filter that can be expressed in one can be expressed in the other. However, in implementation they are very different: The tree model maps naturally into code for a stack machine while the CFG model maps naturally into code for a register machine. Since most modern machines are register based, we will argue that the CFG approach lends itself to a more efficient implementation.

The CSPF (Tree) Model

The CSPF filter engine is based on an operand stack. Instructions either push constants or packet data on the stack, or perform a binary boolean or bitwise operation on the top two elements. A filter program is a sequentially executed list of instructions. After evaluating a program, if the top of stack has a non-zero value or the stack is empty then the packet is accepted, otherwise it is rejected.

There are two implementation shortcomings of the expression tree approach:[7]

- The operand stack must be simulated. On most modern machines this means generating add and subtract operations to maintain a simulated stack pointer and actually doing loads and stores to memory to simulate the stack. Since memory tends to be the major bottleneck in modern architectures, a filter model that can use values in machine registers and avoid this memory traffic will be more efficient.
- The tree model often does unnecessary or redundant computations. For example, the tree in Figure 4 will compute the value of 'ether.type == ARP' even if the test for IP is true. While this problem can be somewhat mitigated by adding 'short circuit' operators to the filter machine, some inefficiency is intrinsic: Because of the hierarchical design of network protocols, packet headers must be

[7]Note that it is not our intention to denigrate CSPF or its enormous contribution to the community – we simply wish to investigate the implementation implications of its filter model when run on modern hardware. The CSPF filtering mechanism was intended to support efficient protocol demultiplexing for user-level network code. The initial implementation achieved huge gains by performing user-specified demultiplexing inside the kernel rather than in a user-process. After this, the incremental gain from a more efficient filter design was negligible and, as a result, the designers of CSPF invested less effort in the filter machinery and, indeed, have pointed out that the "filter language is not a result of careful analysis but rather embodies several accidents of history"[8].

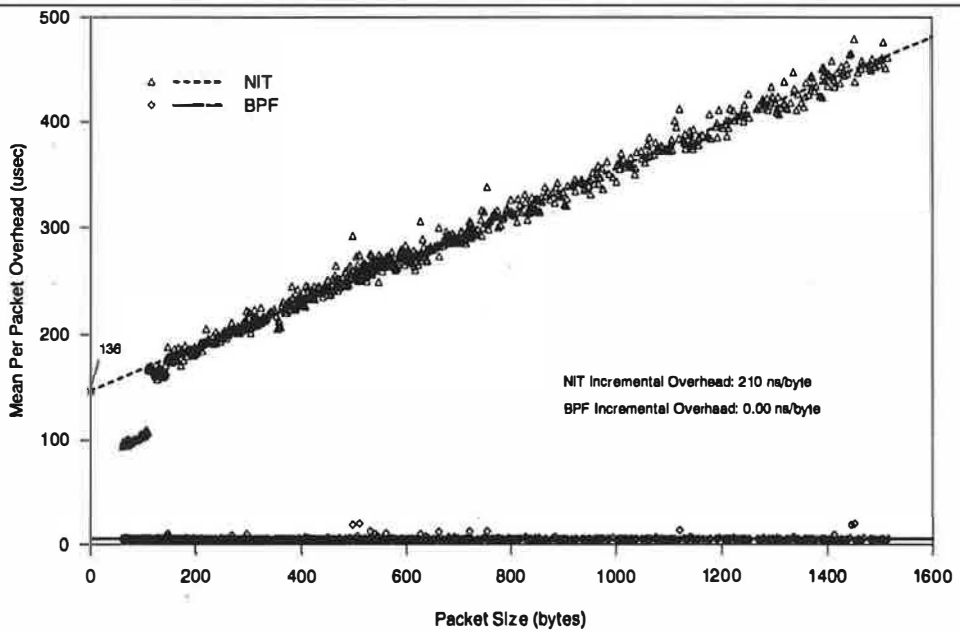


Figure 3: NIT versus BPF: "reject all"

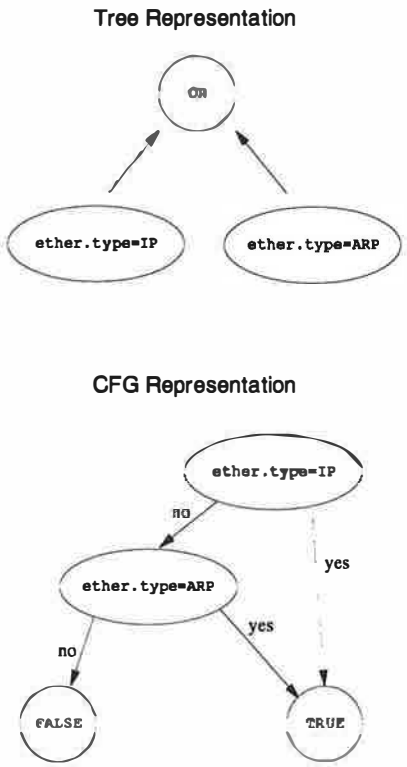


Figure 4: Filter Function Representations

parsed to reach successive layers of encapsulation. Since each leaf of the expression tree represents a packet field, independent of other leaves, redundant parses may be carried out to evaluate the entire tree. In the CFG representation, it is always possible to reorder the graph in such a way that *at most* one parse is done for any layer.[8]

Another problem with CSPF, recognized by the designers, is its inability to parse variable length packet headers, e.g., TCP headers encapsulated in a variable length IP header. Because the CSPF instruction set didn't include an indirection operator, only packet data at fixed offsets is accessible. Also, the CSPF model is restricted to a single sixteen bit data type which results in a doubling of the number of operations to manipulate 32 bit data such as Internet addresses or TCP sequence numbers. Finally, the design does not permit access to the last byte of an odd-length packet.

While the CSPF model has shortcomings, it offers a novel generalization of packet filtering: The idea of putting a pseudo-machine language interpreter in the kernel provides a nice abstraction for describing and implementing the filtering mechanism. And, since CSPF treats a packet as a simple array of bytes, the filtering model is completely protocol independent. (The application that specifies the filter is responsible for encoding the filter appropriately for the underlying network media and protocols.)

The BPF model, described in the next section, is an attempt to maintain the strengths of CSPF while addressing its limitations and the performance shortcomings of the stack-based filter machine.

The BPF Model

CFGs vs. Trees

BPF uses the CFG filter model since it has a significant performance advantage over the expression tree model. While the tree model may need to redundantly parse a packet many times, the CFG model allows parse information to be 'built into' the flow graph. I.e., packet parse state is 'remembered' in the graph since you know what paths you must have traversed to reach to a particular node and once a subexpression is evaluated it need not be recomputed since the control flow graph can always be (re-)organized so the value is only used at nodes that follow the original computation.

For example, Figure 6 shows a CFG filter function that accepts all packets with an Internet address *foo*. We consider a scenario where the network layer protocols are IP, ARP, and Reverse ARP, all of which contain source and destination Internet addresses. The filter should catch all cases.

[8]This graph reordering is, however, a non-trivial problem. Our BPF compiler (part of tcpdump[4]) contains a fairly sophisticated optimizer to reorder and minimize CFG filters. This optimizer is the subject of a future paper.

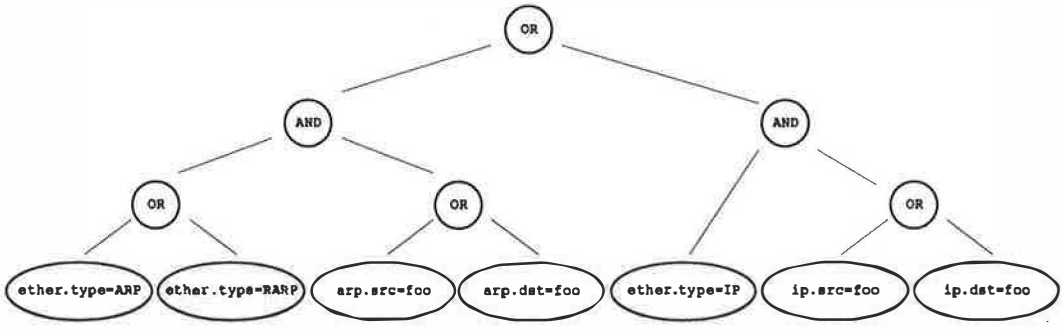


Figure 5: Tree Filter Function for "host foo"

Accordingly, the link layer type field is tested first. In the case of IP packets, the IP host address fields are queried, while in the case of ARP packets, the ARP address fields are used. Note that once we learn that the packet is IP, we do not need to check that it might be ARP or RARP. In the expression tree model, shown in figure 5, seven comparison predicates and six boolean operations are required to traverse the entire tree. The longest path through the CFG has five comparison operations, and the average number of comparisons is three.

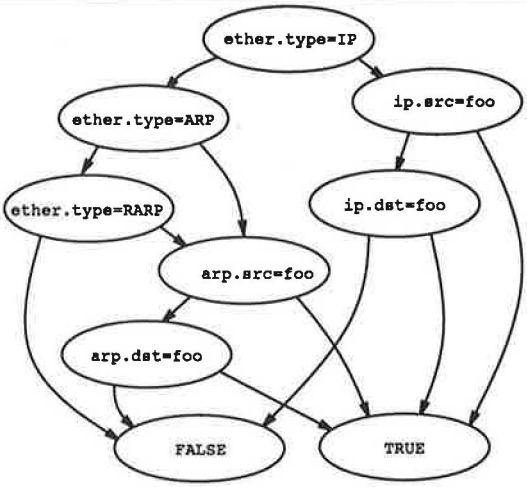


Figure 6: CFG Filter Function for "host foo"

Design of filter pseudo-machine

The use of a control flow graph rather than an expression tree as the theoretical underpinnings of the filter pseudo-machine is a necessary step towards an efficient implementation but it is not sufficient. Even after leveraging off the experience and pseudo-machine models of CSPF and NNStat[1], the BPF model underwent several generations (and several years) of design and test. We believe the current model offers sufficient generality with no sacrifice in performance. Its evolution was guided by the following design constraints:

1. It must be protocol independent. The kernel should not have to be modified to add new protocol support.
2. It must be general. The instruction set should be rich enough to handle unforeseen uses.
3. Packet data references should be minimized.
4. Decoding an instruction should consist of a single C switch statement.
5. The abstract machine registers should reside in physical registers.

Like CSPF, constraint 1 is adhered to simply by not mentioning any protocols in the model. Packets are viewed simply as byte arrays.

Constraint 2 means that we must provide a fairly general computational model, with control flow, sufficient ALU operations, and conventional addressing modes.

Constraint 3 requires that we only ever touch a given packet word once. It is common for a filter to compare a given packet field against a set of values, then compare another field against another set of values, and so on. For example, a filter might match packets addressed to a set of machines, or a set of TCP ports. Ideally, we would like to cache the packet field in a register and compare it across the set of values. If the field is encapsulated in a variable length header, we must parse the outer headers to reach the data. Furthermore, on alignment restricted machines, accessing multi-byte data can involve an expensive byte-by-byte load. Also, for packets in *mbufs*, a field access may involve traversing an *mbuf* chain. After we have done this work once, we should not do it again.

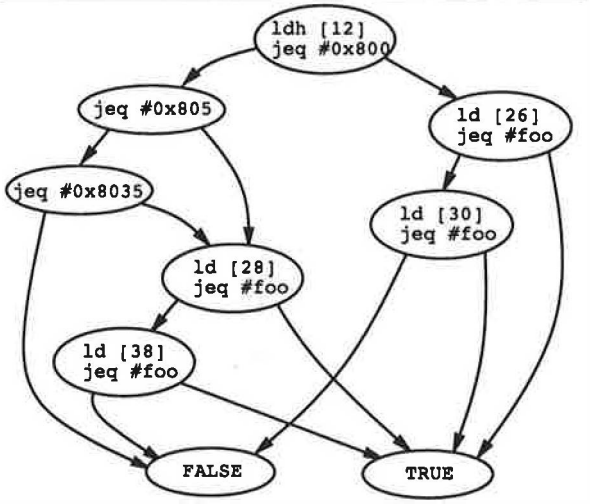


Figure 7: BPF Program for "host foo".

Constraint 4 means that we will have an efficient instruction decoding step but it precludes an orthogonal addressing mode design unless we are willing to accommodate a combinatorial explosion of switch cases. For example, while three address instructions make sense for a real processor (where much work is done in parallel) the sequential execution model of an interpreter means that each address descriptor would have to be decoded serially. A single address instruction format minimizes the decode, while maintaining sufficient generality.

Finally, Constraint 5 is a straightforward performance consideration. Along with constraint 4, it enforces the notion that the pseudo-machine register set should be small.

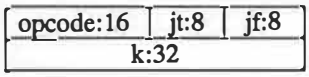
These constraints prompted the adoption of an accumulator machine model. Under this model, each node in the flowgraph computes its corresponding predicate by computing a value into the accumulator and branching based on that value. Figure 7 shows the filter function of Figure 6 using the BPF instruction set.

The BPF Pseudo-Machine

The BPF machine abstraction consists of an accumulator, an index register (x), a scratch memory store, and an implicit program counter. The operations on these elements can be categorized into the following groups:

1. LOAD INSTRUCTIONS copy a value into the accumulator or index register. The source can be an immediate value, packet data at a fixed offset, packet data at a variable offset, the packet length, or the scratch memory store.
2. STORE INSTRUCTIONS copy either the accumulator or index register into the scratch memory store.
3. ALU INSTRUCTIONS perform arithmetic or logic on the accumulator using the index register or a constant as an operand.
4. BRANCH INSTRUCTIONS alter the flow of control, based on comparison test between a constant or x register and the accumulator.
5. RETURN INSTRUCTIONS terminate the filter and indicated what portion of the packet to save. The packet is discarded entirely if the filter returns 0.
6. MISCELLANEOUS INSTRUCTIONS comprise everything else – currently, register transfer instructions.

The fixed-length instruction format is defined by as follows:



The *opcode* field indicates the instruction type and addressing modes. The *jt* and *jf* fields are used by the conditional jump instructions and are the offsets from the next instruction to the true and false targets. The *k* field is a generic field used for various purposes. Table 1 shows the entire BPF instruction set. We have adopted this "assembler syntax" as a means of illustrating BPF filters and for debugging output. The actual encodings are defined with C macros, the details of which we omit here (see [6] for full details). The column labeled *addr modes* lists the addressing modes allowed for each instruction listed in the *opcode* column. The semantics of the addressing modes are listed in Table 2.

The load instructions simply copy the indicated value into the accumulator (`ld`, `ldh`, `ldb`) or index register (`ldx`). The index register cannot use the packet addressing modes. Instead, a packet value must be loaded into the accumulator and transferred to the index register, via `tax`. This is not a common occurrence, as the index register is used primarily to parse the variable length IP header, which can be loaded directly via the `4*([k]&0xf)`

opcodes	*addr modes*				
`ldb`	`[k]`		`[x+k]`		
`ldh`	`[k]`		`[x+k]`		
`ld`	`#k`	`#len`	`M[k]`	`[k]`	`[x+k]`
`ldx`	`#k`	`#len`	`M[k]`	`4*([k]`	
`st`	`M[k]`				
`stx`	`M[k]`				
`jmp`	`L`				
`jeq`	`#k, Lt, Lf`				
`jgt`	`#k, Lt, Lf`				
`jge`	`#k, Lt, Lf`				
`jset`	`#k, Lt, Lf`				
`add`	`#k`		`x`		
`sub`	`#k`		`x`		
`mul`	`#k`		`x`		
`div`	`#k`		`x`		
`and`	`#k`		`x`		
`or`	`#k`		`x`		
`lsh`	`#k`		`x`		
`rsh`	`#k`		`x`		
`ret`	`#k`		`a`		
`tax`					
`txa`					

Table 1: `BPF Instruction Set`

addressing mode. All values are 32 bit words, except packet data can be loaded into the accumulator as unsigned bytes (`ldb`) or unsigned halfwords (`ldh`). Similarly, the scratch memory store is addressed as an array of 32 bit words. The instruction fields are all in host byte order, and the load instructions convert packet data from network order to host order. Any reference to data beyond the end of the packet terminates the filter with a return value of zero (i.e., the packet is discarded).

`#k`	the literal value stored in *k*
`#len`	the length of the packet
`M[k]`	the word at offset *k* in the scratch memory store
`[k]`	the byte, halfword, or word at byte offset *k* in the packet
`[x+k]`	the byte, halfword, or word at offset *x+k* in the packet
`L`	an offset from the current instruction to L
`#k, Lt, Lf`	the offset to `Lt` if the predicate is true, otherwise the offset to `Lf`
`x`	the index register
`4*([k]&0xf)`	four times the value of the low four bits of the byte at offset *k* in the packet

Table 2: BPF Addressing Modes

The ALU operations (`add`, `sub`, etc.) perform the indicated operation using the accumulator and operand, and store the result back into the accumulator. Division by zero terminates the filter.

The jump instructions compare the value in the accumulator with a constant (`jset` performs a "bitwise and" – useful for conditional bit tests). If the result is true (or non-zero), the true branch is taken, otherwise the false branch is taken. Arbitrary comparisons, which are less common, can be done by subtracting and comparing to 0. Note that there are no `jlt`, `jle` or `jne` opcodes since these can be built from the codes above by reversing the branches. Since jump offsets are encoded in eight bits, the longest jump is 256 instructions. Jumps longer than this are conceivable, so a *jump always* opcode is provided (`jmp`) that uses the 32 bit operand field for the offset.

The return instructions terminate the program and indicate how many bytes of the packet to accept. If that amount is 0, the packet will be rejected entirely. The actual amount accepted will be the minimum of the length of the packet and the amount indicated by the filter.

Examples

We now present some examples to illustrate how packet filters can be expressed using the BPF instruction set. (In all the examples that follow, we assume Ethernet format for the link level headers.)

This filter accepts all IP packets:

```
      ldh  [12]
      jeq  #ETHERTYPE_IP, L1, L2
L1:   ret  #TRUE
L2:   ret  #0
```

The first instruction loads the Ethernet type field. We compare this to type IP. If the comparison fails, zero is returned and the packet is rejected. If it is successful, TRUE is returned and the packet is accepted. (TRUE is some non-zero value that represents the number of bytes to save.)

This next filter accepts all IP packets which did not originate from two particular IP networks, 128.3.112 or 128.3.254. If the Ethernet type is IP, the IP source address is loaded and the high 24 bits are masked off. This value is compared with the two network addresses:

```
lll   ldh  [12]
      jeq  #ETHERTYPE_IP, L1, L4
L1:   ld   [26]
      and  #0xffffff00
      jeq  #0x80037000, L4, L2
L2:   jeq  #0x8003fe00, L4, L3
L3:   ret  #TRUE
L4:   ret  #0
```

Parsing Packet Headers

The previous examples assume that the data of interest lie at fixed offsets in the packet. This is not the case, for example, with TCP packets, which are encapsulated in a variable length IP header. The start of TCP header must be computed from the length given in the IP header.

The IP header length is given by the low four bits of the first byte in the IP section (byte 14 on an Ethernet). This value is a word offset, and must be scaled by four to get the corresponding byte offset. The instructions below will load this offset into the accumulator:

```
ldb [14]
and #0xf
lsh #2
```

Once the IP header length is computed, data in the TCP section can be accessed using indirect loads. Note that the effective offset has three components:

- the IP header length,
- the link level header length, and
- the data offset relative to the TCP header.

For example, an Ethernet header is 14 bytes and the destination port in a TCP packet is at byte

two. Thus, adding 16 to the IP header length gives the offset to the TCP destination port. The previous code segment is shown below, augmented to test the TCP destination port against some value *N*:

```
      ldb   [14]
      and   #0xf
      lsh   #2
      tax
      ldh   [x+16]
      jeq   #N, L1, L2
L1:   ret   #TRUE
L2:   ret   #0
```

Because the IP header length calculation is a common operation, the `4*([k]&0xf)` addressing mode was introduced. Substituting in the `ldx` instruction simplifies the filter into:

```
      ldx   4*([14]   0xf)
      ldh   [x+16]
      jeq   #N, L1, L2
L1:   ret   #TRUE
L2:   ret   #0
```

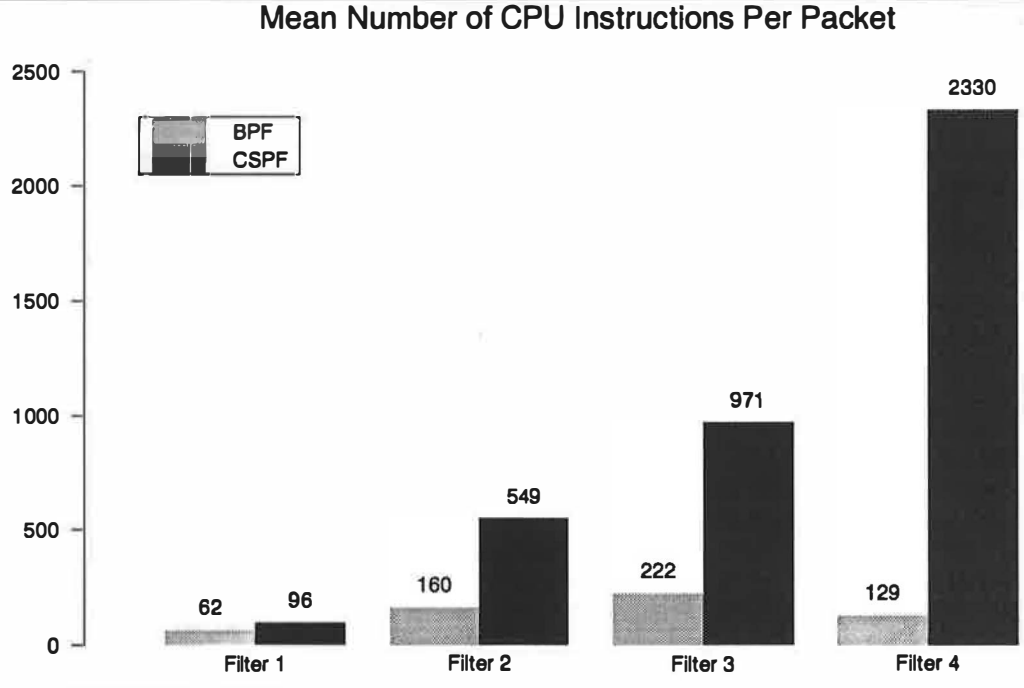
However, the above filter is valid only if the data we are looking at is really a TCP/IP header. Hence, the filter must also check that link layer type is IP, and that the IP protocol type is TCP. Also, the IP layer might fragment a TCP packet, in which case the TCP header is present only in the first fragment. Hence, any packets with a non-zero fragment offset should be rejected. The final filter is shown below:

```
      ldh   [12]
      jeq   #ETHERPROTO_IP, L1, L5
L1:   ldb   [23]
      jeq   #IPPROTO_TCP, L2, L5
L2:   ldh   [20]
      jset  #0x1fff, L5, L3
L3:   ldx   4*([14]   0xf)
      ldh   [x+16]
      jeq   #N, L4, L5
L4:   ret   #TRUE
L5:   ret   #0
```

Filter Performance Measurements

We profiled the BPF and CSPF filtering models outside the kernel using *iprof* [9], an instruction count profiler. To fully compare the two models, an indirection operator was added to CSPF so it could parse IP headers. The change was minor and did not adversely affect the original filtering performance. Tests were run on large packet trace files gathered from a busy UC Berkeley campus network. Figure 8 shows the results for four fairly typical filters.

Filter 1 is trivial. It tests whether one 16 bit word in the packet is a given value. The two



Filter 1	IP packets
Filter 2	IP packets with src or dst "horse"
Filter 3	TCP packets with src or dst port of finger, domain, login, or shell
Filter 4	IP, ARP or RARP packets between hosts "horse" and "gauguin"

Figure 8: BPF/CSPF Filter Performance

models are fairly comparable, with BPF faster by about 50%.

Filter 2 looks for a particular IP host (source or destination) and shows more of a disparity – a performance gap of 240%. The larger difference here is due mostly to the fact that CSPF operates only on 16 bit words and needs two comparison operations to determine the equality of a 32 bit Internet address.

Filter 3 is an example of packet parsing (required to locate the TCP destination port field) and illustrates a yet greater performance gap. The BPF filter parses the packet once, loading the port field into the accumulator then simply does a comparison cascade of the interesting ports. The CSPF filter must re-do the parse and relocate the TCP header for each port to be tested.

Finally, filter 5 demonstrates the effect of the unnecessary computations done by CSPF for a filter similar to the one described in Figures 5 and 6.

Applications

BPF is now about two years old and has been put to work in several applications. The most widely used is *tcpdump* [4], a network monitoring and data acquisition tool. Tcpdump performs three primary tasks: filter translation, packet acquisition, and packet display. Of interest here is the filter translation mechanism. A filter is specified with a user-friendly, high level description language. Tcpdump has a built in compiler (and optimizer) which translates the high level filters into BPF programs. Of course, this translation process is transparent to the user.

Arpwatch [5] is a passive monitoring program that tracks Ethernet to IP address mappings. It notifies the system administrator, via email, when new mappings are established or abnormal behavior is noted. A common administrative nuisance is the use of a single IP address by more than one physical host, which arpwatch dutifully detects and reports.

A very different application of BPF has been its incorporation into a variant of the Icon Programming Language [3]. Two new data types, a *packet* and a *packet generator* have been built into the Icon interpreter. Packets appear as first class record objects, allowing convenient "dot operator" access to packet headers. A packet generator can be instantiated directly off the network, or from a previously collected file of trace data. Icon is an interpreted, dynamically typed language with high level string scanning primitives and rich data structures. With the BPF extensions, it is well suited for the rapid prototyping of networking analysis tools.

Netload and *histo* are two network visualization tools which produce real time network statistics on an X display. Netload graphs utilization data in real time, using tcpdump style filter specifications. Histo produces a dynamic interarrival-time histogram of timestamped multimedia network packets.

The Reverse ARP daemon uses the BPF interface to read and write Reverse ARP requests and replies directly to the local network. (We developed this program to allow us to entirely replace NIT by BPF in our SunOS 4 systems. Each of the Sun NIT-based applications (etherfind, traffic, and rarpd) now has a BPF analog.)

Finally, recent versions of NNStat[1] and nfswatch can be configured to run over BPF (in addition to running over NIT).

Conclusion

BPF has proven to be an efficient, extensible, and portable interface for network monitoring. Our comparison studies have shown that it outperforms NIT in its buffer management and CSPF in its filtering mechanism. Its programmable pseudo-machine model has demonstrated excellent generality and extensibility (all knowledge of particular protocols is factored out of the kernel). Finally, the system is portable and runs on most BSD and BSD-derivative systems[9] and can interact with various data link layers[10].

Availability

BPF is available via anonymous ftp from host `ftp.ee.lbl.gov` as part of the tcpdump distribution, currently in the file `tcpdump-2.2.1.tar.Z`. Eventually we plan to factor BPF out into its own distribution so look for `bpf-*.tar.Z` in the future. Arpwatch and netload are also available from this site.

Acknowledgements

This paper would never have been published without the encouragement of Jeffrey Mogul. Jeff ported *tcpdump* to Ultrix and added little-endian support, uncovering dozens of our byte-ordering bugs. He also inspired the `jset` instruction by forcing us to consider the arduous task of parsing DECNET packet headers. Mike Karels suggested that the filter should decide not only whether to accept a packet, but also how much of it to keep. Craig Leres was the first major user of BPF/tcpdump and is responsible for finding and fixing many bugs in both. Chris Torek helped with the packet processing performance measurements and provided insight on various BSD peculiarities. Finally, we are grateful to the many users and extenders of BPF/tcpdump across the Internet for their suggestions, bug fixes, source code, and the many questions that have, over

[9] SunOS 3.5, HP-300 and HP-700 BSD, SunOS 4.x, 4.3BSD Tahoe/Reno, and 4.4BSD.

[10] Ethernet, FDDI, SLIP, and PPP are currently supported.

the years, greatly broadened our view of the networking world and BPF's place in it.

Finally, we would like to thank Vern Paxson, Craig Leres, Jeff Mogul, Sugih Jamin, and the referees for their helpful comments on drafts of this paper.

Bibliography

[1] Braden, R. T. A pseudo-machine for packet monitoring and statistics. In *Proceedings of SIGCOMM '88* (Stanford, CA, Aug. 1988), ACM.

[2] Digital Equipment Corporation. *packetfilter(4), Ultrix V4.1 Manual.*

[3] Griswold, R. E., and Griswold, M. T. *The Icon Programming Language.* Prentice Hall, Inc., Englewood Cliffs, NJ, 1983.

[4] Jacobson, V., Leres, C., and McCanne, S. *The Tcpdump Manual Page.* Lawrence Berkeley Laboratory, Berkeley, CA, June 1989.

[5] Leres, C. *The Arpwatch Manual Page.* Lawrence Berkeley Laboratory, Berkeley, CA, Sept. 1992.

[6] McCanne, S. *The BPF Manual Page.* Lawrence Berkeley Laboratory, Berkeley, CA, May 1991.

[7] Mogul, J. C. Efficient use of workstations for passive monitoring of local area networks. In *Proceedings of SIGCOMM '90* (Philadelphia, PA, Sept. 1990), ACM.

[8] Mogul, J. C., Rashid, R. F., and Accetta, M. J. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of 11th Symposium on Operating Systems Principles* (Austin, TX, Nov. 1987), ACM, pp. 39--51.

[9] Rice, S. P. *iprof* source code, May 1991. Brown University.

[10] Sun Microsystems Inc. *NIT(4P); SunOS 4.1.1 Reference Manual.* Mountain View, CA, Oct. 1990. Part Number: 800-5480-10.

Author Information

Steven McCanne has been with the Lawrence Berkeley Laboratory since 1988, working on network analysis tools and remote conferencing applications. He holds a B.S. degree in Electrical Engineering and Computer Science from U.C. Berkeley, and is currently a Ph.D. student in Computer Science at U.C.B. His e-mail address is mccanne@ee.lbl.gov.

Van Jacobson's e-mail address is van@ee.lbl.gov.

Reach both authors at: Lawrence Berkeley Laboratory, One Cyclotron Road, Berkeley, CA 94720.

The Organization of Networks in Plan 9

Dave Presotto & Phil Winterbottom – AT&T Bell Laboratories

ABSTRACT

In a distributed system networks are of paramount importance. This paper describes the implementation, design philosophy and organization of network support in Plan 9. Topics include network requirements for distributed systems, our kernel implementation, network naming, user interfaces and performance. We also observe that much of this organization is relevant to current systems.

Introduction

Plan 9 [Pike90] is a general-purpose, multi-user, portable distributed system implemented on a variety of computers and networks. What distinguishes Plan 9 is its organization. The goals of this organization were to reduce administration and to promote resource sharing. One of the keys to its success as a distributed system is the organization and management of its networks.

A Plan 9 system comprises file servers, CPU servers and terminals. The file servers and CPU servers are typically centrally located multiprocessor machines with large memories and high speed interconnects. A variety of workstation-class machines serve as terminals connected to the central servers using several networks and protocols. The architecture of the system demands a hierarchy of network speeds matching the needs of the components. Connections between file servers and CPU servers are high-bandwidth point-to-point fiber links. Connections from the servers fan out to local terminals using medium speed networks such as Ethernet [Met80] and Datakit [Fra80]. Low speed connections via the Internet and the AT&T backbone serve users in Oregon and Illinois. Basic Rate ISDN data service and 9600 baud serial lines provide slow links to users at home.

Since CPU servers and terminals use the same kernel, users may choose to run programs locally on their terminals or remotely on CPU servers. The organization of Plan 9 hides the details of system connectivity allowing both users and administrators to configure their environment to be as distributed or centralized as they wish. Simple commands support the construction of a locally represented namespace spanning many machines and networks. At work, users tend to use their terminals like workstations running interactive programs locally and reserving the CPU servers for data or compute intensive jobs such as compiling and computing chess endgames. At home or when connected over a slow network, users tend to do most work on the CPU server to minimize network traffic. The goal of the network organization is to provide the same environment to the user wherever resources are used.

Kernel Network Support

Networks play a central role in any distributed system. This is particularly true in Plan 9 where most resources are provided by servers external to the kernel. The importance of the networking code within the kernel is reflected by its size; of 25,000 lines of kernel code, 12,500 are network and protocol related. Networks are continually being added and the fraction of code devoted to communications is growing. Moreover, the network code is complex. Protocol implementations consist almost entirely of synchronization and dynamic memory management; areas demanding subtle error recovery strategies. The kernel currently supports Datakit, point-to-point fiber links, an Internet (IP) protocol suite and ISDN data service. The variety of networks and machines has raised issues not addressed by other systems running on commercial hardware supporting only Ethernet or FDDI.

The File System protocol

A central idea in Plan 9 is the representation of a resource as a hierarchical file system. Each process assembles a view of the system by building a *namespace* [Needham] connecting its resources. File systems need not represent disc files; in fact, most Plan 9 file systems have no permanent storage. A typical file system dynamically represents some resource like a set of network connections or the process table. Communication between the kernel, device drivers and local or remote file servers uses a protocol called 9P. The protocol consists of 17 messages describing operations on files and directories. Kernel resident device and protocol drivers use a procedural version of the protocol while external file servers use an RPC form. Nearly all traffic between Plan 9 systems consists of 9P messages. 9P relies on several properties of the underlying transport protocol. It assumes messages arrive reliably and in sequence and that delimiters between messages are preserved. When a protocol does not meet these requirements (for example TCP does not preserve delimiters) we provide mechanisms to marshal messages before handing them to the system.

A kernel data structure, the *channel*, is a handle to a file server. Operations on a channel generate

the following 9P messages. The `auth` and `attach` messages authenticate a connection, established by means external to 9P, and validate its user. The result is an authenticated channel referencing the root of the server. The `clone` message makes a new channel identical to an existing channel, much like the `dup` system call. A channel may be moved to a file on the server using a `walk` message to descend each level in the hierarchy. The `stat` and `wstat` messages read and write the attributes of the file referenced by a channel. The `open` message prepares a channel for subsequent `read` and `write` messages to access the contents of the file. `Create` and `remove` perform the actions implied by their names on the file referenced by the channel. The `clunk` message discards a channel without affecting the file.

A kernel resident file server called the *mount driver* converts the procedural version of 9P into RPC's. The *mount* system call provides a file descriptor, which can be a pipe to a user process or a network connection to a remote machine, to be associated with the mount point. After a mount, operations on the file tree below the mount point are sent as messages to the file server. The mount driver manages buffers, packs and unpacks parameters from messages and demultiplexes among processes using the file server.

Kernel Organization

The network code in the kernel is divided into three layers: hardware interface, protocol processing, and program interface. A device driver typically uses streams to connect the two interface layers. Additional stream modules may be pushed on a device to process protocols. Each device driver is a kernel-resident file system. Simple device drivers serve a single level directory containing just a few files; for example, we represent each UART by a data and a control file.

```
helix% cd /dev
helix% ls -l eia*
--rw-rw-rw- t 0 bootes bootes 0 Jul 16 17:28 eia1
--rw-rw-rw- t 0 bootes bootes 0 Jul 16 17:28 eia1ctl
--rw-rw-rw- t 0 bootes bootes 0 Jul 16 17:28 eia2
--rw-rw-rw- t 0 bootes bootes 0 Jul 16 17:28 eia2ctl
helix%
```

The control file is used to control the device; writing the string *b1200* to `/dev/eialctl` sets the line to 1200 baud.

Multiplexed devices present a more complex interface structure. For example, the LANCE Ethernet driver serves a two level file tree (Figure 1) providing

- device control and configuration
- user-level protocols like *arp*
- diagnostic interfaces for snooping software.

The top directory contains a `clone` file and a directory for each connection, numbered 1 to n. Each connection directory corresponds to an Ethernet packet type. Opening the `clone` file finds an unused connection directory and opens its `ctl` file. Reading the control file returns the ASCII connection number; the user process can use this value to construct the name of the proper connection directory. In each connection directory files named `ctl`, `data`, `stats` and `type` provide access to the connection. Writing the string *connect 2048* to the `ctl` file sets the packet type to 2048 and configures the connection to receive all IP packets sent to the machine. Subsequent reads of the file `type` yield the string *2048*. The `data` file accesses the media; reading it returns the next packet of the selected type. Writing the file queues a packet for transmission after appending a packet header containing the source address and packet type. The `stats` file returns ASCII text containing the interface address, packet input/output counts, error statistics and general information about the state of the interface.

If several connections on an interface are configured for a particular packet type each receives a copy of the incoming packets. The special packet type, `-1`, selects all packets. Writing the strings *promiscuous* and *connect -1* to the `ctl` file configures a conversation to receive all packets on the Ethernet.

Although the driver interface may seem elaborate, the representation of a device as a set of files using ASCII strings for communication has several advantages. Any mechanism supporting remote access to files immediately allows a remote machine

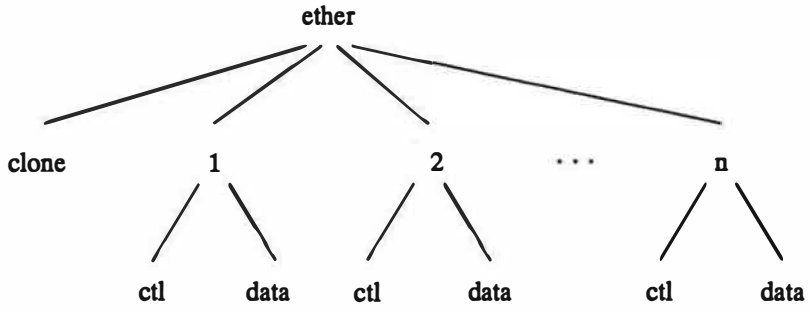


Figure 1: Protocol device driver directory

to use our interfaces as gateways. Using ASCII strings to control the interface avoids byte order problems and ensures a uniform representation for devices on the same machine and even devices accessed remotely. Representing dissimilar devices by the same set of files allows common tools to serve several networks or interfaces. Programs like `stty` are replaced by `echo` and shell redirection.

Protocol devices

Network connections are represented as pseudo-devices called protocol devices. Protocol device drivers exist for the Datakit URP protocol and for each of the Internet IP protocols TCP, UDP, and IL. IL, described below, is a new communication protocol used by Plan 9 for transmitting file system RPC's. All protocol devices look identical so user programs contain no network-specific code.

Each protocol device driver serves a directory structure similar to that of the Ethernet driver. The top directory contains a `clone` file and a directory for each connection numbered `1` to `n` (see Figure 1). Each connection directory contains files to control one connection and to send and receive information. A look at a TCP connection directory provides the output shown in Figure 2.

The files `local`, `remote` and `status` supply information about the state of the connection. The `data` and `ctl` files provide access to the process end of the stream implementing the protocol. The `listen` file is used to accept incoming calls from the network.

The following steps establish a connection.

1. The clone device of the appropriate protocol directory is opened to reserve an unused connection.
2. The file descriptor returned by the open points to the `ctl` file of the new connection. Reading that file descriptor returns an ASCII string containing the connection number.
3. A protocol/network specific ASCII address string is written to the `ctl` file.
4. The path of the `data` file is constructed using the connection number. When the `data` file is opened the connection is established.

A process can read and write this file descriptor to send and receive messages from the network. If the process opens the `listen` file it blocks until an incoming call is received. An address string written to the `ctl` file before the listen selects the ports or services the process is prepared to accept. When an incoming call is received, the open completes and returns a file descriptor pointing to the `ctl` file of the new connection. Reading the `ctl` file yields a connection number used to construct the path of the `data` file. A connection remains established while any of the files in the connection directory are referenced or until a close is received from the network.

Streams

A *stream* [Rit84a][Presotto] is a bidirectional channel connecting a physical or pseudo-device to user processes. The user processes insert and remove data at one end of the stream. Kernel processes acting on behalf of a device insert data at the other end. Asynchronous communications channels such as pipes, TCP conversations, Datakit conversations, and RS232 lines are implemented using streams.

A stream comprises a linear list of *processing modules*. Each module has both an upstream (toward the process) and downstream (toward the device) *put routine*. Calling the put routine of the module on either end of the stream inserts data into the stream. Each module calls the succeeding one to send data up or down the stream.

An instance of a processing module is represented by a pair of *Queues*, one for each direction. The queues point to the put procedures and can be used to queue information traveling along the stream. Some put routines queue data locally and send it along the stream at some later time either due to a subsequent call or an asynchronous event such as a retransmission timer or a device interrupt.

```
helix% cd /net/tcp/2
helix% ls -l
--rw-rw---- I 0 ehg     bootes 0 Jul 13 21:14 ctl
--rw-rw---- I 0 ehg     bootes 0 Jul 13 21:14 data
--rw-rw---- I 0 ehg     bootes 0 Jul 13 21:14 listen
--r--r--r-- I 0 bootes  bootes 0 Jul 13 21:14 local
--r--r--r-- I 0 bootes  bootes 0 Jul 13 21:14 remote
--r--r--r-- I 0 bootes  bootes 0 Jul 13 21:14 status
helix% cat local remote status
135.104.9.31 5012
135.104.53.11 564
tcp/2 1 Established connect
helix%
```

Figure 2: TCP connection directory

Processing modules create helper kernel processes to provide a context for handling asynchronous events. For example, a helper kernel process awakens periodically to perform any necessary TCP retransmissions. The use of kernel processes instead of serialized run-to-completion service routines differs from the implementation of Unix streams. Unix service routines cannot use any blocking kernel resource and they lack a local long-lived state. Helper kernel processes solve these problems and simplify the stream code.

There is no implicit synchronization in our streams. Each processing module must ensure that concurrent processes using the stream are synchronized. This maximizes concurrency but introduces the possibility of deadlock. However deadlocks are easily avoided by careful programming; to date they have not caused us problems.

Information is represented by linked lists of kernel structures called *blocks*. Each block contains a type, some state flags, and pointers to an optional buffer. Block buffers can hold either data or control information, i.e., directives to the processing modules. Blocks and block buffers are dynamically allocated from kernel memory.

User Interface

A stream is represented at user level as two files, `ctl` and `data`. The actual names can be changed by the device driver using the stream, as we saw earlier in the example of the UART driver. The first process to open either file creates the stream automatically. The last close destroys it. Writing to the `data` file copies the data into kernel blocks and passes them to the downstream put routine of the first processing module. A write of less than 32K is guaranteed to be contained by a single block. Concurrent writes to the same stream are not synchronized although the 32K block size assures atomic writes for most protocols. The last block written is flagged with a delimiter to alert downstream modules that care about write boundaries. In most cases the first put routine calls the second, the second calls the third, and so on until the data is output. As a consequence most data is output without context switching.

Reading from the `data` file returns data queued at the top of the stream. The read terminates when the read count is reached or when the end of a delimited block is encountered. A per stream read lock ensures only one process can read from a stream at a time and guarantees that the bytes read were contiguous bytes from the stream.

Like UNIX streams Plan 9 streams can be dynamically configured. The stream system intercepts and interprets the following control blocks:

push *name* adds an instance of the processing module *name* to the top of the stream.

pop removes the top module of the stream.

hangup sends a hangup message up the stream from the device end.

Other control blocks are module-specific and are interpreted by each processing module as they pass.

The convoluted syntax and semantics of the UNIX `ioctl` system call convinced us to leave it out of Plan 9. Instead, `ioctl` is replaced by the `ctl` file. Writing to the `ctl` file is identical to writing to a `data` file except the blocks are of type *control*. A processing module parses each control block it sees. Commands in control blocks are ASCII strings, so byte ordering is not an issue when one system controls streams in a name space implemented on another processor. The time to parse control blocks is not important, since control operations are rare.

Device Interface

The module at the downstream end of the stream is part of a device interface. The particulars of the interface vary with the device. Most device interfaces consist of an interrupt routine, an output put routine, and a kernel process. The output put routine stages data for the device and starts the device if it is stopped. The interrupt routine wakes up the kernel process whenever the device has input to be processed or needs more output staged. The kernel process puts information up the stream or stages more data for output. The division of labor among the different pieces varies depending on how much must be done at interrupt level. However, the interrupt routine may not allocate blocks or call a put routine since both actions require a process context.

Multiplexing

The conversations using a protocol device must be multiplexed onto a single physical wire. We push a multiplexor processing module onto the physical device stream to group the conversations. The device end modules on the conversations add the necessary header onto downstream messages and then put them to the module downstream of the multiplexor. The multiplexing module looks at each message moving up its stream and puts it to the correct conversation stream after stripping the header controlling the demultiplexing.

This is similar to the Unix implementation of multiplexor streams. The major difference is that we have no general structure that corresponds to a multiplexor. Each attempt to produce a generalized multiplexor created a more complicated structure and underlined the basic difficulty of generalzing this mechanism. We now code each multiplexor from scratch and favour simplicity over generality.

Reflections

Despite five year's experience and the efforts of many programmers, we remain dissatisfied with the stream mechanism. Performance is not an issue; the

time to process protocols and drive device interfaces continues to dwarf the time spent allocating, freeing, and moving blocks of data. However the mechanism remains inordinately complex. Much of the complexity results from our efforts to make streams dynamically configurable, to reuse processing modules on different devices and to provide kernel synchronization to ensure data structures don't disappear under foot. This is particularly irritating since we seldom use these properties.

Streams remain in our kernel because we are unable to devise a better alternative. Larry Peterson's X-kernel [Pet89a] is the closest contender but doesn't offer enough advantage to switch. If we were to rewrite the streams code, we would probably statically allocate resources for a large fixed number of conversations and burn memory in favor of less complexity.

The IL Protocol

None of the standard IP protocols are suitable for transmission of 9P messages over an Ethernet or the Internet. TCP has a high overhead and does not preserve delimiters. UDP, while cheap, does not provide reliable sequenced delivery. Early versions of the system used a custom protocol that was efficient but unsatisfactory for internetwork transmission. When we implemented IP, TCP and UDP we looked around for a suitable replacement with the following properties:

- Reliable datagram service with sequenced delivery
- Runs over IP
- Low complexity, high performance
- Adaptive timeouts

None met our needs so a new protocol was designed. IL is a lightweight protocol designed to be encapsulated by IP. It is a connection-based protocol providing reliable transmission of sequenced messages between machines. No provision is made for flow control since the protocol is designed to transport RPC messages between client and server. A small outstanding message window prevents too many incoming messages from being buffered; messages outside the window are discarded and must be retransmitted. Connection setup uses a two way handshake to generate initial sequence numbers at each end of the connection; subsequent data messages increment the sequence numbers allowing the receiver to resequence out of order messages. In contrast to other protocols, IL does not do blind retransmission. If a message is lost and a timeout occurs, a query message is sent. The query message is a small control message containing the current sequence numbers as seen by the sender. The receiver responds to a query by retransmitting missing messages. This allows the protocol to behave well in congested networks, where blind retransmission would cause further congestion. Like TCP, IL has adaptive timeouts. A round-trip timer is used to calculate acknowledge and retransmission times in terms of the network speed. This allows the protocol to perform well on both the Internet and on local Ethernets.

In keeping with the minimalist design of the rest of the kernel, IL is small. The entire protocol is 847 lines of code, compared to 2200 lines for TCP. IL is our protocol of choice.

Network Addressing

A uniform interface to protocols and devices is not sufficient to support the transparency we require. Since each network uses a different addressing scheme, the ASCII strings written to a control file have no common format. As a result, every tool must know the specifics of the networks it is capable of addressing. Moreover, since each machine supplies a subset of the available networks, each user must be aware of the networks supported by every terminal and server machine. This is obviously unacceptable.

Several possible solutions were considered and rejected; one deserves more discussion. We could have used a user-level file server to represent the network namespace as a Plan 9 file tree. This global naming scheme has been implemented in other distributed systems. The file hierarchy provides paths to directories representing network domains. Each directory contains files representing the names of the machines in that domain; an example might be the path `/net/name/usa/edu/mit/ai`. Each machine file contains information like the IP address of the machine. We rejected this representation for several reasons. First, it is hard to devise a hierarchy encompassing all representations of the various network addressing schemes in a uniform manner. Datakit and Ethernet address strings have nothing in common. Second, the address of a machine is often only a small part of the information required to connect to a service on the machine. For example, the IP protocols require symbolic service names to be mapped into numeric port numbers, some of which are privileged and hence special. Information of this sort is hard to represent in terms of file operations. Finally, the size and number of the networks being represented burdens users with an unacceptably large amount of information about the organization of the network and its connectivity. In this case the Plan 9 representation of a resource as a file is not appropriate.

If tools are to be network independent, a third-party server must resolve network names. A server on each machine, with local knowledge, can select the best network for any particular destination machine or service. Since the network devices present a common interface, the only operation which differs between networks is name resolution. A symbolic name must be translated to the path of

the clone file of a protocol device and an ASCII address string to write to the `ctl` file. A connection server (CS) provides this service.

Network Database

On most systems several files in the `/etc` directory (such as `hosts`, `networks`, `services`, `hosts.equiv`, `bootptab`, and `named.d`) hold network information. Much time and effort is spent administering these files and keeping them mutually consistent. Tools attempt to automatically derive one or more of the files from information in other files but maintenance continues to be difficult and error prone.

Since we wrote our world from scratch, we decided to avoid this nightmare. One database on a shared server contains all the information needed for network administration. Two ASCII files comprise the main database: `/lib/ndb/local` contains locally administered information and `/lib/ndb/global` contains information imported from elsewhere. The files contain sets of attribute/value pairs of the form *attr=value*, where *attr* and *value* are alphanumeric strings. Systems are described by multi-line entries; a header line at the left margin begins each entry followed by zero or more indented attribute/value pairs specifying names, addresses, properties, etc. For example, the entry for our CPU server specifies a domain name, an IP address, an Ethernet address, a Datakit address, a boot file, and supported protocols.

```
sys = helix
    dom=helix.research.att.com
    bootf=/mips/9power
    ip=135.104.9.31 ether=0800690222f0
    dk=nj/astro/helix
    proto=il flavor=9cpu
```

If several systems share entries such as network mask and gateway, we specify that information with the network or subnetwork instead of the system. The following entries define a class B IP network and a few subnets derived from it. The entry for the network specifies the IP mask, file system, and authentication server for all systems on the network. Each subnetwork specifies its default IP gateway.

```
ipnet=mh-astro-net ip=135.104.0.0
    ipmask=255.255.255.0
    fs=bootes.research.att.com
    auth=1127auth
ipnet=unix-room ip=135.104.117.0
    ipgw=135.104.117.1
ipnet=third-floor ip=135.104.51.0
    ipgw=135.104.51.1
ipnet=fourth-floor ip=135.104.52.0
    ipgw=135.104.52.1
```

Database entries also define the mapping of service names to port numbers for TCP, UDP, and IL:

```
tcp=echo        port=7
tcp=discard     port=9
tcp=systat      port=11
tcp=daytime     port=13
```

All programs read the database directly so there are no intermediate files or binary format and consistency problems are rare. However the database files can become large. Our global file, containing all information about both Datakit and Internet systems in AT&T, has 43,000 lines. To speed searches, we build hash table files for each attribute we expect to search often. The hash file entries point to entries in the master files. Every hash file contains the modification time of its master file so we can avoid using an out-of-date hash table. Searches for attributes that aren't hashed or whose hash table is out-of-date still work, they just take longer.

Connection Server

On each system a user level connection server process, CS, performs symbolic name to address translation. CS uses information about available networks, the network database, and other servers (such as DNS) to translate names. CS is a file server serving a single file, `/net/cs`. A client writes a symbolic name to `/net/cs` then reads one line for each matching destination reachable from this system. The lines are of the form *filename message*, where *filename* is the path of the clone file to open for a new connection and *message* is the string to write to it to make the connection. The following example illustrates this. `Ndb/csquery` is a program that prompts for strings to write to /net/cs and prints the replies.

```
% ndb/csquery
> net!helix!9fs
/net/il/clone 135.104.9.31!17008
/net/dk/clone nj/astro/helix!9fs
```

CS provides meta-name translation to perform complicated searches. The special network name `net` selects any network in common between source and destination supporting the specified service. A host name of the form *$attr* is the name of an attribute in the network database. The database search returns the value of the matching attribute/value pair most closely associated with the source host. Most closely associated is defined on a per network basis. For example, the symbolic name *tcp!$auth!rexauth* causes CS to search for the *auth* attribute in the database entry for the source system, then its subnetwork (if there is one) and then its network.

```
% ndb/csquery
> net!$auth!rexauth
/net/il/clone 135.104.9.34!17021
/net/dk/clone nj/astro/p9auth!rexauth
/net/il/clone 135.104.9.6!17021
/net/dk/clone nj/astro/musca!rexauth
```

Normally CS derives naming information from its database files. For domain names however, CS first consults another user level process, the domain name server (DNS). If no DNS is reachable, CS relies on its own tables.

Like CS, the domain name server is a user level process providing one file, `/net/dns`. A client writes a request of the form *domain-name type*, where *type* is a domain name service resource record type. DNS performs a recursive query through the Internet domain name system producing one line per resource record found. The client reads `/net/dns` to retrieve the records. Like other domain name servers, DNS caches information learned from the network. DNS is implemented as a multi-process shared memory application with separate processes listening for network and local requests.

Library routines

The section on protocol devices described the details of making and receiving connections across a network. The dance is straightforward but tedious. Library routines are provided to relieve the programmer of the details.

Connecting

The *dial*(2) library call establishes a connection to a remote destination. It returns an open file descriptor for the `data` file in the connection directory.

```
int dial(char *dest,
    char *local, char *dir, int *cfdp)
```

dest is the symbolic name/address of the destination.

local is the local address. Since most networks do not support this, it is almost always zero.

dir is a pointer to a buffer to hold the path name of the protocol directory representing this connection. *Dial* fills this buffer if the pointer is non-zero.

cfdp is a pointer to a file descriptor for the `ctl` file of the connection. If the pointer is non-zero, `dial` opens the control file and tucks the file descriptor here.

Most programs call *dial* with a destination name and all other arguments zero. *Dial* uses CS to translate the symbolic name to all possible destination addresses and attempts to connect to each in turn until one works. Specifying the special name `net` in the network portion of the destination allows CS to pick a network/protocol in common with the destination for which the requested service is valid. For example, assume the system *research.att.com* has the Datakit address *nj/astro/research* and IP addresses *135.104.117.5* and *129.11.4.1*. The call

```
fd = dial("net!research.att.com!login",
                        0, 0, 0, 0);
```

tries in succession to connect to *nj/astro/research/login* on the Datakit and both *135.104.117.5!513* and *129.11.4.1!513* across the Internet.

Dial accepts addresses instead of symbolic names. For example, the destinations *tcp!135.104.117.5!513* and *tcp!research.att.com!login* are equivalent references to the same machine.

Listening

A program uses four routines to listen for incoming connections. It first *announce*(2)'s its intention to receive connections, then *listen*(2)'s for calls and finally *accept*(2)'s or *reject*(2)'s them. *Announce* returns an open file descriptor for the `ctl` file of a connection and fills *dir* with the path of the protocol directory for the announcement.

```
int  announce(char *addr, char *dir)
```

Addr is the symbolic name/address announced; if it does not contain a service, the announcement is for all services not explicitly announced. Thus, one can easily write the equivalent of the `inetd` program without having to announce each separate service. An announcement remains in force until the control file is closed.

Listen returns an open file descriptor for the `ctl` file and fills *ldir* with the path of the protocol directory for the received connection. It is passed *dir* from the announcement.

```
int  listen(char *dir, char *ldir)
```

Accept and *reject* are called with the control file descriptor and *ldir* returned by *listen*. Some networks such as Datakit accept a reason for a rejection; networks such as IP ignore the third argument.

```
int  accept(int ctl, char *ldir)
int  reject(int ctl, char *ldir,
                        char *reason)
```

The code in Figure 2 implements a typical TCP listener. It announces itself, listens for connections and forks a new process for each. The new process echoes data on the connection until the remote end closes it. The `"*"` in the symbolic name means the announcement is valid for any addresses bound to the machine the program is run on.

User Level

Communication between Plan 9 machines is done almost exclusively in terms of 9P messages. Only two services *cpu* and *exportfs* are used. The *cpu* service is analogous to `rlogin`. However, rather than emulating a terminal session across the network, *cpu* creates a process on the remote machine whose namespace is an analogue of the window in which it was invoked. *Exportfs* is a user

level file server which allows a piece of namespace to be exported from machine to machine across a network. It is used by the *cpu* command to serve the files in the terminal's namespace when they are accessed from the cpu server.

By convention the protocol and device driver file systems are mounted in a directory called `/net`. Although the per-process namespace allows users to configure an arbitrary view of the system, in practice most user profiles build a conventional namespace.

Exportfs

Exportfs is invoked by an incoming network call. The `listener` (the Plan 9 equivalent of `inetd`) runs the profile of the user requesting the service to construct a namespace before starting *exportfs*. After an initial protocol establishes the root of the file tree being exported, the remote process mounts the connection allowing *exportfs* to act as a relay file server. Operations in the imported file tree are executed on the remote server and the results returned. As a result the namespace of the remote machine appears to be exported into a local file tree.

The *import* command calls *exportfs* on a remote machine, mounts the result in the local namespace and then exits. No process is required locally to serve mounts; 9P messages are generated by the mount driver in the kernel and sent directly over the network.

Exportfs must be multithreaded since the system calls *open*, *read* and *write* may block. Plan 9 does not implement the *select* system call but does

```
int
echo_server(void)
{
        int dfd, lcfd;
        char adir[40], ldir[40];
        int n;
        char buf[256];

        afd = announce("tcp!*!echo", adir);
        if(afd < 0)
                return -1;

        for(;;){
                /* listen for a call */
                lcfd = listen(adir, ldir);
                if(lcfd < 0)
                        return -1;

                /* fork a process to echo */
                switch(fork()){
                case 0:
                        /* accept the call and open the data file */
                        dfd = accept(lcfd, ldir);
                        if(dfd < 0)
                                return -1;

                        /* echo until EOF */
                        while((n = read(dfd, buf, sizeof(buf))) > 0)
                                write(dfd, buf, n);
                        exits(0);
                case -1:
                        perror("forking");
                default:
                        close(lcfd);
                        break;
                }
        }
}
```

Figure 2: Typical TCP listener

allow processes to share file descriptors, memory and other resources. *Exportfs* and the configurable namespace provide a powerful means of sharing resources between machines. It is a building block for constructing complex namespaces served from many machines.

The simplicity of the interfaces encourages naive users to exploit the potential of a richly connected environment. Using these tools it is easy to gateway between networks. For example on a terminal with only a Datakit connection:

```
import -a helix /net
telnet ai.mit.edu
```

The *import* command makes a Datakit connection to the machine helix where it starts an instance *exportfs* to serve `/net`. The *import* command mounts the remote `/net` directory after (the *-a* option to *import*) the existing contents of the local `/net` directory. The directory contains the union of the local and remote contents of `/net`. Local entries supersede remote ones of the same name so networks on the local machine are chosen in preference to those supplied remotely. However, unique entries in the remote directory are now visible in the local `/net` directory. The networks helix supports over and above the Datakit are available to the terminal. The effect on the namespace is shown by the following example:

```
philw-gnot% ls /net
/net/cs
/net/dk
philw-gnot% import -a musca /net
philw-gnot% ls /net
/net/cs
/net/cs
/net/dk
/net/dk
/net/dns
/net/ether
/net/il
/net/tcp
/net/udp
```

Ftpfs

Fed up with the user unfriendly interface of the ftp command on other systems, we decided to make our ftp interface a file system. Our command, *ftpfs*, dials the ftp port of a remote system, prompts for login and password, sets image mode, and mounts the remote file system onto `/n/ftp`. Files and directories are cached to reduce traffic. The cache is updated whenever a file is created. Ftpfs works with TOPS-20, VMS, and various Unix flavors as the remote system.

Cyclone Fiber Links

The file servers and CPU servers are connected by high-bandwidth point-to-point links. A link consists of two VME cards connected by a pair of optical fibers. The VME cards use 33Mhz Intel 960 processors and AMD's TAXI fiber transmitter/receivers to drive the lines at 125 Mbit/sec. Software in the VME card reduces latency by copying messages from system memory to fiber without intermediate buffering.

Performance

We've measured both latency and throughput of reading and writing bytes between two processes for a number of different paths. Measurements were made on two and four CPU SGI Power Series processors. The CPU's are 25 MHZ MIPS 3000's. The latency is measured as the round trip time for a byte sent from one process to another and back again. Throughput is measured using 16k writes from one process to another.

Table 1 – Performance

test	throughput MBytes/sec	latency millisec
pipes	8.15	.255
IL/ether	1.02	1.42
URP/datakit	0.22	1.75
Cyclone	3.2	0.375

Conclusion

The representation of all resources as file systems coupled with an ASCII interface has proved more powerful than we had originally imagined. Resources can be used by any computer in our networks independent of byte ordering or CPU type. The connection server provides an elegant means of decoupling tools from the networks they use. Users successfully use Plan 9 without knowing the topology of the system or the networks they use. More information about 9P can be found in the Plan 9 Programmers manual available by anonymous ftp from research.att.com in the directory dist/pla9doc.

References

[Pike90] R. Pike, D. Presotto, K. Thompson, H. Trickey, "Plan 9 from Bell Labs", *UKUUG Proc. of the Summer 1990 Conf., London, England, 1990*

[Needham] R. Needham, "Names", in Distributed systems, S. Mullender, ed., Addison Wesley, 1989

[Presotto] D. Presotto, "Multiprocessor Streams for Plan 9", *UKUUG Proc. of the Summer 1990 Conf.,* London, England, 1990

[Met80] R. Metcalfe, D. Boggs, C. Crane, E. Taf and J. Hupp, "The Ethernet Local Network: Three reports", *CSL-80-2,* XEROX Palo Alto Research Center, February 1980.

[Fra80] A. G. Fraser, "Datakit - A Modular Network for Synchronous and Asynchronous Traffic",

Proc. International Conference on Communication, Boston Mass., June 1980.

[Pet89a] L. Peterson, "RPC in the X-Kernel: Evaluating new Design Techniques", *Proc. Twelfth Symposium on Operating Systems Principles*, Litchfield Park AZ, December 1990.

[Rit84a] D. M. Ritchie, "A Stream Input-Output System", *AT&T Bell Laboratories Technical Journal, 68(8)*, October 1984.

Author Information

Phil Winterbottom's email address is philw@research.att.com.

Removable Media in Solaris

Howard Alt – SunSoft, Incorporated

ABSTRACT

Since the dawn of time (or at least Jan 1, 1970) it has been difficult for the common user to take advantage of removable media under UNIX. The traditional UNIX approach to dealing with removable media has been to let programs name the device containing the media, and to leave it to the operator (or user) to ensure that the right media is in the device named. We have implemented the opposite approach: having the program specify the media and letting the OS take care of the device. Media is referenced by a name in the file system and recognized when it is inserted into a device. Administrators may specify actions to be taken when media is named, recognized, and removed.

Introduction

The UNIX interface to removable media (floppys, tapes, etc) has traditionally been a minimalist one. The user specifies the physical device in order to gain access to their media. There is no assurance that the expected media is in the device, and the only security is per-device. In addition, there is no general interface for applications to recognize the insertion of media.

Under UNIX, in order to gain access to a file system on a floppy, for example, a user must become root, know what file system type is on the floppy, and execute the mount command (e.g., mount -F pcfs /dev/fd0c /mnt). This is quite a lot for a user to know, especially the users that most workstation companies are trying to attract these days.

In MS-DOS, when a floppy is inserted it is instantly accessible as A: (or B:). The user is not required to know anything about mounting or file systems. The only way to reference a disk, however, is by drive name.

On the Macintosh, users are not forced to deal with drive names, and they do not have to know about file system type. Users of the Macintosh deal with media names, rather than devices.

The UNIX model makes it very difficult to layer applications that use the various forms of removable media. Users of removable media find UNIX systems very difficult to use.

Media Management

Goals

Our overall direction has been to create a high level model which will scale to a wide range of removable media, and to provide extensibility at several levels. In effect, we wanted to build a platform on which customers and third parties can build easy to use applications.

We have set forth some high level goals which have guided many of our architectural and design decisions.

Provide an abstraction for the media

Applications and users should not refer to a device, instead they should be provided a name to refer to the media.

This is a fundamental change in the existing access model, but it is required to implement any general solution. We have found this to be the biggest stumbling block for people who are used to the "old way".

Security

Security should be maintained on a per-media basis, rather than a per-device bases.

Users "own" media, not devices. Protecting the drive from read or write access creates only problems. When media has the notion of an "owner", normal UNIX access semantics can be applied per-media.

Insertion/ejection paradigm

There should be a uniform interface for programs to recognize the insertion or ejection of any type of media.

It should be possible to easily implement programs which interact with and take advantage of removable media.

Operator interface

Since applications (or users) may reference media that is not in a drive, an interface should be provided to notify operators of media requests.

Extensible device and label interface

We cannot predict all the types of devices that users will want to connect to our machines, or the types of media that may exist. Interfaces must exist which customers or vendors can extend the product.

Design Considerations

What Media Should be Supported

Since this mechanism needs to be useful for a broad range of removable media, we had to be careful to not design *out* any particular type of media. We decided to accomplish this by not designing *in*

any particular form of removable media. To provide this level of abstraction, we have moved the details of managing devices and interpreting labels out into dynamic shared objects which are loaded into the user-level media manager at run time. The core of the media manager is only aware of some very basic properties of the devices and labels it is managing.

Operator Considerations

Each type of media is potentially used differently. For example, in general, floppies are directly attached to the workstation being used. Magneto-optical disks, however, are generally contained in an autochanger. The mechanism for notifying an "operator" is driven by a configuration file, which allows for directing messages relating to the floppy to the screen and sending an e-mail message to an operator for magneto-optical media.

Media Names

A natural UNIX way to implement a new name space is a file system. We have chosen this method to present new block and character special devices that represent media.

For some time, there has been an implementation of a user-level NFS server. It was a fairly trivial matter to take this server and implement our own file system. The user-level NFS server has the advantage of being easy to debug, and all of the interfaces are very well defined.

Database

Since we've implemented this name space in terms of a user-level NFS server, we need some way to keep changes to the name space across reboots, and perhaps share the name space between machines. A conflicting goal is for the media manager to "just work" out of the box, with no explicit set-up. We have experimented with two different databases. One that just stores the data in-memory, and another which stores the data in NIS+[2].

The in-memory database is not persistent across reboots, but requires no special configuration. The NIS+ database requires special configuration. We are currently shipping with the in-memory database as the default. In the future, we will be providing a file based database, and we will also be publishing the interface for customers and third partys to craft their own ways to store and share the name space.

Devices over the network

An opportunity we considered was to provide a mechanism for users and applications to access devices on different machines, over the network. The model would be that it doesn't matter which machine your chosen media is on, reads and writes to the name would just get to the right place.

At some level, this would be a nice feature to have. The reason we choose not to implement it at this time is because of the complexity involved in matching up ioctls between different machine architectures and flavors of UNIX.

Other Work

There are many implementations of "labeled" media access for various operating systems, including UNIX. To our knowledge, all of these require linking with a special library and many require the use of new interfaces. In addition, these implementations frequently support only one type of media.

We decided early on that our support for removable media must use existing interfaces, and that we needed to be able to support a broad range of media types and devices. We also wanted an implementation that would require very little special support from existing device drivers.

Architecture

The media manager is implemented as a kernel driver, and a user level daemon, see Figure 1. The daemon is a user-level NFS server, which automatically mounts a file system on "/vol". The daemon presents names of different media in this name space. Names for the media are block and character special devices. The major number of these devices is that of the vol driver, and the minor number is assigned per virtual media. The vol driver is loaded (via an ioctl) with mappings between the "virtual" minor number and a physical device.

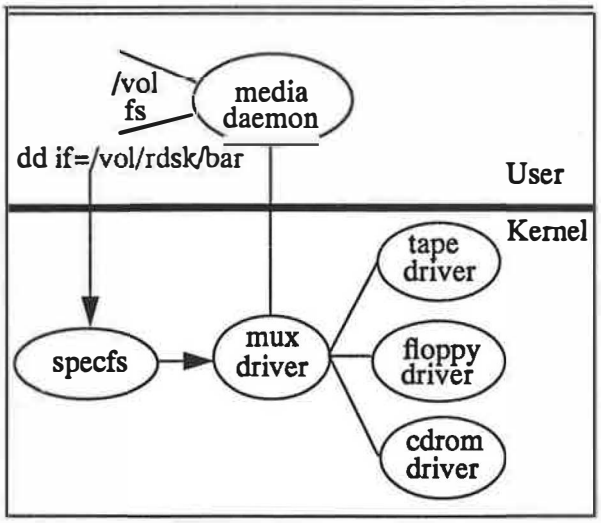


Figure 1: Media manager

If a media name is opened and the driver does not contain a mapping for that media, the daemon is notified. At this point, the daemon uses a database to discover where it might be, or might have been, checks the suspected device for the requested media, and loads the mapping into the driver. If the daemon is unable to locate the media, a "notify" event (see below) is generated and someone is told about the problem.

The daemon is controlled by a configuration file, which specifies which devices are being

managed, what sort of media can be expected in a given device, and which labels can be expected on given media. In addition, programs are specified to be executed when certain events happen.

The daemon uses threads [5] to provide concurrency between accesses to the /vol name space, and label checking operations. The daemon is implemented in such a way as to be portable between different threads implementations, making minimum necessary use of the concurrency mechanisms.

Name Space

The name space, rooted at /vol, provides access to media names and allows the simple perusal and modification of the names. The file system interface is a very effective tool for presenting and manipulating these names.

In the /vol name space, file system operations work as expected, with the exception that regular files may not be created, and special files may not be explicitly created (via mknod). Other file system operations such as ls, mv, mkdir, chmod, chown, chgrp, ln, and rm work, for the most part, as expected.

The name space consists of a physical portion and a logical portion. The logical part of the name space represents media names independent of their location. A user (or application) can access a piece of media without knowledge of where it is. The "system" can be relied on to provide the correct media at some point in the future (assuming operators are awake and such). The physical part of the name space is provided to allow access to media in a specific drive. For example, media that is unlabeled cannot have a name in the logical part of the name space, but can be accessed through the physical name space.

The name space allows access to both the block and character special interfaces. The logical part of the name space has three directorys: /vol/dsk, /vol/rdsk, and /vol/rmt. /vol/dsk and /vol/rdsk contain the block and character names (respectively) for random access media (disks). /vol/rmt contains the character names for sequential access media (tapes).

The physical part of the name space is contained under /vol/dev. The hierarchy under /vol/dev is intended to mimic /dev to a certain degree. The floppy drive is represented as /vol/dev/fd0, and /vol/dev/rfd0. Media that is inserted in the floppy drive would appear as /vol/dev/fd0/frog, assuming one inserted a floppy named "frog" in fd0. In addition, there are symbolic links that are automatically built to allow the construction of simple programs. The floppy symbolic link would be /vol/dev/aliases/floppy0 -> /vol/dev/rfd0/frog. The "floppy0" alias comes from the configuration file (see below).

Some forms of media support partitions. If a piece of media has partitions, its name in the name space becomes a directory, with the special devices appearing under it. The special devices are named using the SVR4 convention: s0, s1, etc. For example, the "toad" CD-ROM might be represented as /vol/rdsk/toad/s0, and /vol/rdsk/toad/s2.

Tapes devices frequently support several different access methods and densities. Tapes are represented as a directory with the access methods represented as files under the directory. A tape named "foo" would have the following access methods: /vol/rmt/foo/d, /vol/rmt/foo/n, /vol/rmt/foo/b, /vol/rmt/foo/bn. The "d" node would yield default System V semantics, "n" is default System V semantics with no-rewind on close, "b" is Berkeley mode, and "bn" is Berkeley with no-rewind on close.

Before a tape has a label, it can be written at any density. There are four densities supported: low, medium, high, and ultra (l, m, h, u). All the permutations of access methods and densities are available on unlabeled tapes. Once a tape has a label, its density is selected and cannot be written at a different density unless the label is "scratched".

Labels

Media is expected to have some sort of label, for normal named access to work. Label formats are specified to the media manager in the form of a shared object (e.g., label_dos.so, Figure 2) with several entry points used to identify and interpret the label. The interface to these shared objects will be published and supported so that customers and third parties can provide support for their own labels. The number of lines of C code to implement support for a label type is around 400.

```
# Labels supported
label dos label_dos.so floppy
label cdrom label_cdrom.so cdrom
label sun label_sun.so floppy
```

Figure 2: Excerpt from /etc/vold.conf

Not all media is labeled, in particular media that is fresh out of a box, or media that contains data for exchange between machines without labeled access. For this purpose, a special name is provided in the /vol/dev portion of the name space. For example, a floppy which has a cpio or tar file on it would be available as /vol/dev/fd0/unlabeled. A floppy fresh out of the box would be available as /vol/dev/fd0/unformatted. The distinction between unlabeled and unformatted is that unformatted is not readable by the drive, and is assumed to not have any data on it. Unlabeled, on the other hand has data on it, the format of the label is simply not recognizable.

The media manager only looks at labels, not file systems. For example, it is possible to have a floppy that is unlabeled and still have a file system on it. One way to get such a floppy is to write a tar file out to it, then run newfs on it.

Code currently exists to interpret Sun labels, DOS labels, and CD-ROM labels. The label code can try to look at the label and guess at a good name for that media. If the media manager has never "seen" that piece of media before, it will request a name, owner, group, file modes, and other things from the label code. Existing label drivers are only able to derive a name, however future labels (or custom ones) might keep more information.

CD-ROM labels are an interesting case because the standards for labels on them are very weak. There is no requirement that they be unique in any way. For example, two completely different CD-ROM's may have exactly the same Sun label. The same is true, although less likely, with High Sierra. The solution we've chosen is to assume that the first 64K bytes of data on the CD-ROM is unique. We choose that size because the root directories of UNIX file systems, and High Sierra file systems fall in this range. The root directories (and super blocks) contain time stamps and other unique data. A 128 bit digital signature is then generated from this 64K of data, using the "RSA Data Security, Inc. MD4 Message-Digest Algorithm" [3, 4]. This fast algorithm produces a digital signature which is adequate for our identification purposes.

Devices

Devices that the media manager knows how to deal with are represented by a shared object, much like a label (Figure 3). The code that manages a device must initialize the device, build some data structures, and be able to generate a message when new media appears or is ejected. Normally, these "device drivers" create a thread which listens for insertion or ejection events in some device dependent way. Device management code is roughly the same size as most label code, about 400 lines.

The media manager uses existing device drivers with very little modification. In fact, all the prototype work was carried out with no changes to the drivers. The one change that we have made is an ioctl which blocks, waiting for media to be inserted or ejected. This keeps the daemon (actually, the shared object that supports the device) from having to poll the device open routine to see if there is new media.

Databases

The media manager maintains information about each piece of media, like the name, owner, group, and so on, in a "database". This database provides for sharing the name space over the network, or simply keeping it local. Since there is such a wide variety of data management requirements out there, we've chosen to open up this interface as well.

Attributes

Each piece of media has attributes associated with it. The attributes are things like the type of the label, its owner, the number of partitions, and so on. These are system defined attributes. There is also a generalized interface to provide users or applications the ability to set their own attributes. For example on an audio CD, an audio player program might choose to keep the title of tracks in the database. These attributes are manipulated with library functions and are simply ascii "attribute=value" pairs, much like shell environment variables. Attributes are stored in the database, along with other information about the media.

Insert, Eject, and Notify Events

A configuration file allows the specification of a program to be run when new media arrives, media ejection is requested, or when media is referenced but isn't in a drive. These are called insert, eject, and notify events respectively.

Figure 4 shows the specification of insert, eject, and notify events. Each event keyword is followed by a regular expression (sh style) which specifies which names in the name space will trigger a particular event. A series of flags follow, and use the

```
# Devices to use
use cdrom drive /dev/dsk/c0t6 dev_cdrom.so cdrom0
use floppy drive /dev/fd0 dev_floppy.so floppy0
```

Figure 3: Excerpt from /etc/vold.conf

```
# Events
insert /vol/dev/fd[0-9]/* user=root /usr/sbin/diskovery -D
insert /vol/dev/dsk/* user=root /usr/sbin/diskovery -D
eject /vol/dev/fd[0-9]/* user=root /usr/sbin/diskovery -D
eject /vol/dev/dsk/* user=root /usr/sbin/diskovery -D
notify /vol/rdsk/* group=tty /usr/lib/vold/volmissing -c
```

Figure 4: Excerpt from /etc/vold.conf

keyword=value paradigm. Finally, an event line specifies a program to run when the specified event occurs. Each event program is run, by default, with user=daemon and group=other. The diskovery program [1] (see below) must be root, because it needs to mount file systems. The volmissing program must be in the tty group because it needs permission to write to the console.

In the case of the diskovery program, the -D flag turns on debugging messages to the console.

Eject events are unique because the event is generated after the user has typed the eject(1) command, and before the device is actually ejected. This allows the application to clean up things, sync the file system (or unmount it), or whatever. The exit code for the program is also examined, and if it returns 1, the ejection is denied (EBUSY is returned to the eject ioctl(2)).

Experience

Automatic Mounting of CD-ROM and Floppy

For several years, Sun customers have been requesting a mechanism by which they can mount media like CD-ROM's and floppies without having to become root. On top of the media manager, we have implemented "diskovery".

Diskovery is executed when CD-ROM's and floppies are inserted or ejected. On insertion, it will discover the file system type, and attempt to mount the partition. On ejection, it will unmount the file system.

```
# File system identification
ident hsfs ident_hsfs.so cdrom
ident ufs ident_ufs.so cdrom floppy
ident pcfs ident_pcfs.so floppy
```

Figure 5: Excerpt from /etc/diskovery.conf

Determining the file system type is performed by a function that is kept in a dynamic shared object. This "ident" function decides if the type is right, and also lets the upper level know if it is "clean". A configuration file (Figure 5) specifies which file systems are appropriate to which media, so that long searches can be avoided. The interface to the shared objects will be published so customers or third parties can provide support for their own file system types. The number of lines in these modules average about 60, most of which is boiler plate.

Dirty file systems are cleaned before mounting, or they are mounted read-only if they are not cleanable.

Mounts are performed with the "nosuid" option, which keeps users from carrying around a floppy or CD-ROM's with set-uid programs on them. In addition, the nosuid semantic has been extended to mean that access to block and character special devices is disallowed. This keeps a user from building a file system with devices like memory or disks to which they have privileged access.

Floppies are mounted via the name /floppy/<media_name>; CD-ROM's are mounted as /cdrom/<media_name>. If a floppy or CD-ROM has partitions, they are each mounted (e.g., /cdrom/solaris_2_0/s0, /cdrom/solaris_2_0/s2, etc.).

After mounting (or unmounting) the file system(s), a list of "actions" are run (Figure 6). These actions can do things like notify the file manager program that new media has arrived. In the case of a CD-ROM, an action is provided to check the media for audio tracks and execute the "workman" program.

Diskovery implements a policy which requires that a file system be unmounted before it can be ejected. The media manager does not require this to be the case, however it greatly simplifies the most common user model to do this.

For the future we are considering adding the ability to specify that inserted media be automatically exported upon insertion. An extension to this would be to use the automounter to make this media available in a "known" place all over the network. Another use of the automounter would be to allow it to mount the file systems locally.

Magneto-optical Autochanger

We have experimented with a magneto-optical autochanger, to convince ourselves that the implementation scales beyond the simple CD-ROM and floppy on a workstation. One difference between an autochanger and one or two drives is that looking at each disk after a reboot is very time consuming, about 15 minutes for a full autochanger. If the database is persistent, it will remember which slot each disk was in, and not check until it needs that disk. If the disk is not found in the "remembered" slot, it is searched for in the autochanger and other drives that could contain that media type.

For the magneto-optical autochanger, we chose to have the driver provide a full autochanger

```
# Actions
action cdrom action_filemgr.so
action floppy action_filemgr.so
action cdrom action_workman.so /home/rmtc/halt/bin/workman
```

Figure 6: Excerpt from /etc/diskovery.conf

interface. Each slot in the autochanger looks to the outside world as a separate device. Access to the devices are scheduled using algorithms implemented in the driver.

Tapes

We are still early in the development of our media manager support of tapes. Tapes, of course, are sequential access devices and require a slightly different model in some areas. For example, the media manager cannot simply read the tape label after it's written without waiting until the device is no longer busy.

One of the biggest problems with tapes is that they degrade over time. Often this isn't discovered until the desired data can't be read. Tape drives commonly keep all sorts of statistics about error rates. These statistics are almost never collected. Collecting statistics about how much a drive or tape has been used, and what sort of error rates are being seen is a critical part of our tape support.

Currently, there are several tape autochangers available. One way to implement support for these devices is to provide an interface like we did with the magneto-optical autochanger. In other words, hide the autochanger mechanics in the kernel driver. Another way to implement support for these is to build the selection and scheduling of media in to the media manager driver, and use ioctls to "pick" the media that is placed in a drive. We are currently investigating ways to provide "generic" autochanger support using this method.

Performance

There are several ways to characterize the performance of the media manager. The most interesting is the amount of time it takes between insertion of media in a drive and the time the insert program is executed. This is the most critical for CD-ROM and floppy recognition. Unfortunately, this also poses the biggest problem.

The CD-ROM drive on the SPARCstation is a SCSI device. Being a SCSI device, it is limited to responding to querys, rather than initiating action. In particular, we must ask the device every so often if it has a new piece of media, or if the media has been removed from it. In addition, it takes about 3 seconds for it to scan the surface of the disk upon media insertion. We add another two seconds on top of that to read data off the disk so a unique signature can be generated. If the CD-ROM is being automatically mounted, the act of mounting takes around one second per-partition. We are working along several paths to improve the performance.

Like the CD-ROM drive, the floppy drive doesn't generate interrupts, it just responds to querys. The good news, however, is that once the floppy is detected, reading the label is fairly fast.

The time to eject a CD-ROM or floppy is primarily gated by the speed of fork/exec and umount(2). Currently around two seconds, the time depends on how many partitions were mounted.

The next set of interesting performance metrics is read and write. In particular, does the media manager generate any overhead during normal read and write operations. We have measured the throughput of the vol driver, using a data generator on one end, and dd(1) on the other and the effect of having the vol driver interposed was not measurable.

Operations on names in the /vol name space (e.g., mv, rm, ln, ...) are viewed to be infrequent, and hence their performance is not critical to us. Our performance goal for this was to have operations be on par with NFS operations between like machines. The speed of these operations is affected by which type of database is in use. For the "in memory" database, we exceed the goal by a wide margin. For the experimental NIS+ database, we meet the goal.

Availability

Binary in Solaris

We are delivering the media manager into Solaris in phases, starting with minimal functionality required for the automatic mounting of CD-ROM's and floppies. This will be delivered in a release in the near future. The next phase will include support for tape devices and will have all the interfaces documented.

Source

For support of new devices or labels on machines running Solaris, example code for devices and labels can be provided at no cost.

The product should be fairly portable to other UNIX implementations that support threads. Inquiries for full source to this product are welcome.

References

[1] Alt, H., "CD-ROM's and Floppies in Solaris", *Proc. Jan 1993 UK UNIX User Group/Sun UK User Group Conference*, January, 1993.

[2] McManis, Chuck, "Naming Systems: A Replacement for NIS", *Proc. September 1991 Sun UK User Group Conference*, September, 1991.

[3] Rivest, R., "The MD4 message digest algorithm", in A.J. Menezes and S.A. Vanstone, editors, Advances in Cryptology - CRYPTO '90 Proceedings, pages 303-311, Springer-Verlag, 1991.

[4] Rivest, R., "The MD4 Message Digest Algorithm", RFC 1186, MIT, October 1990.

[5] D. Stein, D. Shah, "Implementing Lightweight Threads", Proc. 1992 USENIX Summer Conference, June, 1992.

Author Information

Howard started his career at a startup, administering their machines. He then went to SRI International to do networking and kernel development. Next, he went to Sun Microsystems to do kernel porting for a few years. After getting tired of the bay area, he returned to Austin to work for Tandem on various kernel development projects. Howard currently works for SunSoft's Rocky Mountain Technology Center in the Storage Management group. Howard can be reached at (719) 528-4614, or by mail at halt@central.sun.com.

An Advanced Tape Cataloging System for UNIX Systems

Christopher J. Calabrese – AT&T Bell Laboratories

ABSTRACT

The problem of tracking large numbers of computer tapes has been largely neglected in the UNIX environment. Commercial UNIX tape cataloging systems are neither as powerful as those for proprietary mainframe and minicomputer operating systems, nor do they fully embrace the UNIX tool-building philosophy.

This paper describes the tape cataloging system now in use at the AT&T Bell Laboratories Homer Computer Center, where UNIX systems have been used for large-scale data processing since the mid 1980's. This system provides the flexible tool-building environment and file-naming semantics that UNIX is famous for, while also providing all the information needed to track thousands of tapes. The need for user training has been minimized by making the tape-specific commands similar to standard UNIX commands. In addition to the user level interface, there is an interface for automated tape mounting services.

Major technical features of the system internals are a UNIX-like filesystem inside a relational database, a UNIX-like programming interface to this filesystem, with UNIX utilities ported to it, and transparent networking.

The Problem With UNIX Tape Handling

In the early days of UNIX, tapes were used for backups and for loading new OS releases. Because those early systems needed only simple tape handling mechanisms, today's UNIX systems have serious deficiencies for tape-intensive data processing.

These deficiencies aren't only in the device drivers. There is a complete lack of standard software to deal with tracking and cataloging tapes, as well as a lack of software for automatic tape mounting. This may change in the future (there are certainly vendors who want their products to be the standard), but for now things are fairly jumbled.

In 1992, the Homer Computer Center will use around 20,000 tapes for backups, and receive around 15,000 tapes from outside sources (that's 7 tera-bytes of tape turnover per year). We need high-quality tape software.

In the past few years, the Advanced Computing Group, which runs the Homer Computer Center, has been aggressively attacking these tape handling problems. On the hardware side, the center has recently purchased a StorageTek NearLine 4400 Automated Cartridge System (a couple of robotic arms, 6,000 IBM 3480 tape cartridges, and some very fast tape drives).

On the software side, we've developed a tape drive allocator[1], a tape cataloging system (the software described in this document), and an automated tape mounting system that interfaces with the StorageTek robot. As well, the mounting system will soon interface with robots and carousels for 8mm and 4mm tape cartridges and with human operators.

Requirements for a Tape Cataloging System

After looking at commercial tape cataloging systems, we decided that they had inherited the early UNIX bias toward only using tapes as backup media, while our needs were for high performance tape-based data processing. We were also concerned that these tools were monolithic and didn't follow the UNIX tool-building philosophy of using flexible, interconnected components. Here is a list of basic requirements we compared against the commercial systems and later used to guide our own development efforts:

- Track All Tape Data
  This includes who owns the tape, where it's permanently stored, where it's currently located, who's currently using it, who has permission to use it, what is stored on it, and when it should be discarded.
- Track All Dataset Information
  In a certain sense, tape is just another storage medium. Just as the filesystem abstraction is used for partitioning disks, the dataset abstraction is used to group sets of tapes (or pieces of them in the general case) together into logical files. Just as users are really interested in disk-files, not disk-blocks, they are interested in datasets, not tapes.
- Easy to Access
  Many cataloging systems (including the one previously used in the Homer Computer Center) index data by owner and a cryptic

name. A flexible naming scheme like the UNIX directory structure is needed.

- Automatic Tape Registration
  The system must be able to accept a batch of labeled tapes (the kind that come from non-UNIX mainframes and minicomputers) from a user and automatically register them into the system. To do this, it must read the tape labels, figure out what they mean, and register both the tapes and the datasets in the database.
- Network Transparent
  The system must be available from any host on the network. The system will never be popular if users (and administrators) have to *rlogin* to a special server to access it.
- Hooks for an Automated Tape Mounting Service
- Works With Old Programs
  The system shouldn't require any programs that read or write tapes to do something special. This would break too many useful programs like *dump* and *tar*.
- Easy to Use
- Easy to Administer
- Robust

Commercial Tape Cataloging Systems

The commercial systems we looked at scored high points on tracking the kind of information about the tapes and datasets that we need, on network transparency, and on robustness.

They did poorly on making data easy to find, usually having an arrangement like a flat namespace for each user. They also did poorly on automating tape registration, not quite meeting our requirement of being able to throw a batch of tapes at the system and have it do the rest. Although I admit that this is one of our toughest requirements to meet, it's also among the most important.

On interfacing with automatic tape mounting services, the systems allowed add-on packages to be bought to interface with specific robotic hardware from major manufactures, but we also need the flexibility to interface with other products like robots for 8mm tape drives.

On ease of use and ease of administration, if you equate ease of use with graphical user interfaces, the commercial systems do well in this respect. If you equate it with leveraging the knowledge you already have about dealing with data abstractions (through the UNIX tools) or with being able to take the reports the system generates about the tapes you own and doing interesting stuff with the data in *sh*, *awk*, and *perl*, they fall pretty short here.

Finally, these systems tend to force programs to either use their supplied libraries for accessing the tapes or to interact with tapes by piping data to/from a supplied program. Neither solution is useful for dealing with vendor-supplied programs that access raw tape devices (i.e., *dump*). Neither solution is particularly efficient. We want our DMA.

Actually, the fact that the products deal directly with tapes and tape robots is evidence of their monolithic approach. Our approach (which we feel is both more flexible and easier to manage) has one component for allocating tape drives, one for dealing with robots and operators, and one for cataloging tapes. Dealing with the tapes themselves is a separate issue (no special libraries needed). We also question design issues like having database systems integrated into the products. While we understand the need in a commercial environment, we'd rather let the database design experts worry about it than invent our own.

Fundamental Design Concepts

The underlying organization of catalog data in this system has great impact on what goes on in the higher levels.

One of our earliest decisions was to store the data in a filesystem-like structure. One reason was to get the flexible naming semantics of a directory structure, but there are also strong similarities between the data needed for this system and the data provided in a UNIX filesystem:

- file/tape ownership
- group ownership
- access permissions
- access/create/update times

Another borrowed filesystem concept deals with tracking data blocks. The inode entries for a disk-file tell the system where the blocks holding the data are found. A conceptually similar structure is used to tell our system where the pieces of a tape-dataset are found.

Data Object Types

There are three main data object types, and each has a corresponding file-type in the filesystem.

- **Tapes.** Tape files keep track of the physical tapes. They are similar to device files for disk-drives.
  Tapes have "special" information associated with them not found in a UNIX filesystem - things like the external tape label, the type of tape, where the tape is physically located, and whether it's currently in use.
- **Storage Objects.** Storage objects hold physical tapes. In the database, storage objects track how many tapes and of what type the physical storage objects can store. They also provide a way to specify where tapes are located.
- **Datasets.** Datasets are represented in the system as Lists Of Tapes (LOTs), which tell

the system how to string physical tape files together into logical files (datasets). This is conceptionally similar to disk-files, which string disk-blocks together.

Less fundamental data types are stored outside the filesystem, including a table of tape media types, a table of tape formats, and audit and accounting logs.

How To Organize the Data

After considering a special filesystem type (too complicated for the foundation of the project, maybe something for a later date), storing data in regular disk-files (no atomic transactions), and a part-filesystem part-database hybrid (too difficult to coordinate the two pieces), we decided to store our data in a filesystem-like structure inside a commercial relational database.

Accessing The Data

Once we had a firm idea of how data was to be organized we turned our thoughts toward how to access it:

- UNIX Utilities
  With data that looks like a UNIX-filesystem, it was natural to access it with the standard UNIX utilities.
- UNIX Source Code
  The system's application programming interface (API) was made to look like the UNIX API, allowing us to port standard utilities to the system. We used BSD 4.3 Tahoe as the source base.
- Networking based on the *rsh*[2] Model
  Since we already had a model for transparent networking to run UNIX utilities, we used it.

Implementing A Filesystem In A Database

This section presents a sketch of how the database filesystem works.

Directories

Taken individually, UNIX directories are essentially relational tables. To store the entire directory tree in a single relational table, however, some changes must be made to the traditional structure. The solution used in this system is to group the members of a directory by that directory's inode number. This is similar to the directory structure of the Macintosh Hierarchical Filesystem[3].

Inode Table

To model a traditional UNIX filesystem, we would need an additional relation to hold inode information. By not allowing hard-links, however, the system combines the directory and inode data into a single structure. This is also a feature of the Plan 9 filesystem[4].

As for the part of the inode that points to the file's disk blocks, this directory/inode structure eliminates the need for that piece altogether for directories. Other file types have auxiliary relations specific to the file type.

This type of database/filesystem organization can be useful when the filesytem data contains information different from that in a real filesystem, when the transactional capabilities of a database are needed, or when the data are to be accessed in ways other than through the directory structure (our system falls into all these categories). Such a filesystem was envisioned by Barry Shein several years ago[5], and similar methods have been used in a distributed filesystem for Plan 9[6].

Implementation Details

This section of the paper describes the system's implementation, starting with the user interface, and ending with the API.

User and Administrator Interface

The interface for users and administrators is an extension of the standard UNIX command interface consisting of a set of shell-level commands such as *tcd*, *tpwd*, *tchmod*, *trm*, *tls*, and *tfsck*.

tcd and *tpwd* are shell functions that operate on the environment variable `TPWD`. Most other utilities are implemented as calls to *tsh*, which handles the network connections. Some administration tools, like *tfsck*, are stand-alone.

A Taste of the Tools

Here's a taste of a typical interaction with the system:

Change directories...

```
$ tcd /home/cjc/data/BLD
```

Look around...

```
$ tls -l
total 15
-rw-r--r--  1 cjc      bld-grp
  16 May 14 15:12 BLD.FEB.92
-rw-r--r--  1 cjc      bld-grp
  13 May 14 15:12 BLD.JAN.92
```

Get information on one of the files...

```
$ tlist BLD.FEB.92
-------- BLD.FEB.92 --------
owner       cjc
group       bld-grp
  ---- tape file 1
  storage   /store/nearline
  tape id   504973
  type      3480
[output truncated for brevity...]
```

Allocate a tape drive...

```
$ alloc -t 3480
hera:/dev/rimt/0
```

Inform the tape mounting system that we want to read this LOT on this drive...

```
$ tmount BLD.FEB.92 /dev/rimt/0
```

Read the tapes (we can use tnext *to change tapes if the program doesn't know how)...*

```
$ suitable-program /dev/rimt/0
```

Clean up...

```
$ tdismount /dev/rimt/0
$ dealloc -f /dev/rimt/0
```

Networking with *tsh*

When *tsh* is called, it opens a socket on a privileged port to the database server machine and connects to a server (*tshd*) using `rexec()`[7]. The code for *tsh* and *tshd* is similar to the BSD 4.3 Tahoe code for *rsh* and *rshd*[8]. Some of the things that *tsh* and *tshd* do differently from *rsh* and *rshd* are:

- No Login Restrictions
  We have a uniform set of user and group id's on our systems. We block accounts on specific machines by giving them a dummy shell (`/etc/fakesh`). *tsh* ignores the shell and allows access from any valid account.
- Proper Return Values
  Exit codes are sent on the *stderr* stream with the simple protocol of a 0 byte followed by a return code byte. This allows scripts using *tsh* to detect errors, something I often wish *rsh* allowed.
- Not Using `exec()`
  Since we're just running a single server on the remote side, *tshd* doesn't have to `exec()` anything. Instead, a miniature shell supporting only built-in commands is called to execute the command.
- Proper Argument Passing
  - Space characters are quoted with a backslash.
  - Backslashes are quoted with a backslash.
  - Arguments are separated by a space character.

  Once again, it would have been nice if *rsh* had gotten this right.

The Programming Interface

At the core of the general API for this system is a set of functions for dealing with database transactions and connections. The next layer is similar to the UNIX system call interface. The final layer deals with manipulating the various object types (tapes, storage objects, and datasets/LOTs). There are create, delete, modify, and several selection functions for each object type.

A simplified API is provided for access by systems providing automatic tape mounting services.

Current System Usage

Although enhancements are still being made to the automatic tape mounting system, the cataloging system gets heavy usage. There are over 12,000 tapes currently in the system, and it grows by about 1,000 per month. Both the users and the system administrators have been happy with it.

Future Directions

Much of the future depends on how well the system stands up to the full load envisioned of 100,000 tapes and on how much demand there is for this system outside the Homer Computer Center.

Performance Enhancements

Current performance levels are quite good (though they already represent a couple of generations of tweaks), but more radical data-caching methods may be employed in the future.

Database Portability

The current system relies heavily on the INGRES database system. If the system is to be used in other locations, it could be made database-independent.

Making the API More UNIX-like

The current API has been optimized to take advantage of the way data are stored and accessed in this system. As a result, some UNIX utilities, including *ls* and *find* had to be "ported" to this system to attain reasonable performance levels. Better caching may alleviate this problem.

A Real Filesystem

Given an API with nearly-complete UNIX filesystem semantics, it should be fairly easy to build a server that allows the database to be mounted as an NFS (or its moral equivalent) filesystem.

Conclusions

The UNIX filesystem and utilities are powerful and familiar. Database technology is equally powerful, if not as familiar. The combination of these two ideas has made for a very powerful and easy to use system. This is the best kind of software re-use.

Software Availability

Inquiries about availability of this software, of the resource allocator, of the tape mounting software, and about other tape handling software developed by the Advanced Computing Group

should be made to Ann Martin (electronically at: arm@ulysses.att.com, or paper at AT&T Bell Laboratories; 600 Mountain Avenue; Murray Hill, NJ 07974).

References

[1] G. G. Smith, "A Distributed Resource Allocator for UNIX systems," *Proceedings of Summer 1989 USENIX Conference*, PP. 95-108, Baltimore, MD, June 12-16, 1989.

[2] Regents of the University of California, *UNIX User's Reference Manual*, University of California at Berkeley, 1984.

[3] Apple Computer, *Inside Macintosh*, Addison-Wesley, 1988.

[4] R. Pike, D. Presotto, K. Thompson, and H. Trickey, "Plan 9 from Bell Labs," AT&T Bell Laboratories, 1990.

[5] B. Shein, Private Communication, 1992.

[6] J. Korn, A. Hume, Private Communication, 1991-1992.

[7] Regents of the University of California, *UNIX Programmer's Reference Manual*, University of California at Berkeley, 1984.

[8] Regents of the University of California, *UNIX System Manager's Manual*, University of California at Berkeley, 1984.

Author Information

Chris Calabrese works in the Advanced Computing Group of AT&T Bell Laboratories in Murray Hill, NJ, where he spends most of his time as a system administrator, programmer, and hacker. He is also pursuing his MS/CS at New York University's Courant Institute. Reach him via U.S. Mail at AT&T Bell Laboratories; 600 Mountain Avenue; Murray Hill, NJ 07974. Reach him electronically at cjc@ulysses.att.com.

Efficient Kernel Memory Allocation on Shared-Memory Multiprocessors

Paul E. McKenney & Jack Slingwine – Sequent Computer Systems, Inc.

ABSTRACT

There has been great progress from the traditional allocation algorithms designed for small memories to more modern algorithms exemplified by McKusick's and Karel's allocator[7]. Nonetheless, none of these algorithms have been designed to meet the needs of UNIX kernels supporting commercial data-processing applications in a shared-memory multiprocessor environment.

On a shared-memory multiprocessor, memory is a global resource. Therefore, allocator performance depends on synchronization primitives and manipulation of shared data as well as on raw CPU speed.

Synchronization primitives and access to shared data depend on system bus interactions. The speed of system busses has not kept pace with that of CPUs, as witnessed by the ever-larger caches found on recent systems. Thus, the performance of synchronization primitives and of memory allocators that use them have not received the full benefit of increased CPU performance.

This situation calls for a new approach to global memory allocation that is not so dependent on synchronization primitives and manipulation of shared data. This paper presents such an approach, which exhibits near-linear speedup on multiprocessors as well as fifteen times the performance of the traditional algorithm when run on a single CPU. Nonetheless, this allocator presents an interface identical to the standard System V UNIX allocator and performs the efficient online coalescing required by many commercial data-processing environments.

Introduction

Parallel implementations of UNIX have been quite successful at meeting the the needs of online transaction-processing (OLTP) applications. Nonetheless, one weakness of previous implementations has been the general-purpose kernel memory allocator.

The old version of the allocator is a straightforward global allocator whose critical sections are protected by spinlocks. Although this worked quite well on older platforms, this allocator's performance is less than optimal on newer platforms, primarily because the speed of synchronization primitives (such as spinlocks) has not increased as rapidly as the speed of other instructions.

There has also been great progress in the area of multiprocessor synchronization primitives (see Herlihy [1] for an overview and references). However, synchronization requires global processing. Global processing is very costly in comparison to local processing and can be expected to become even more expensive as technology advances [2, 10]. We therefore decided to abandon the search for ever-more sophisticated synchronization primitives in favor of a search for an algorithm that does not depend so heavily on synchronization. This search bore fruit in the form of an algorithm that runs fifteen times as fast as the old allocator on a single processor and that exhibits linear speedup on shared-memory multiprocessors, resulting in more than a three-orders-of-magnitude increase in performance, while adding online coalescing.

The next Section analyzes the behavior of the old algorithm. Subsequent sections present the new algorithm and its evaluation.

Analysis

Our investigation into kernel-memory-allocation performance began when we found that the STREAMS[9] buffer allocator was running four to five times more slowly than predicted by instruction counts. We quickly realized that the general-purpose kernel-memory allocator suffered from the same problem, which motivated us to develop the algorithm described in this paper.

The remainder of this section presents the results of the investigation, describing the initial behavior of *allocb* (the STREAMS buffer allocator) and *freeb* (the STREAMS buffer deallocator) and showing how the current allocation algorithm's interaction with the shared-memory multiprocessor environment leads to this behavior. All measurements presented in this section were taken on a Sequent S2000/200 with a pair of 25 MHz 80486

CPUs running DYNIX/ptx, a parallel variant of UNIX.

Behavior of allocb

The *allocb* function returns a pointer to a message, which consists of a message block, data block, and STREAMS buffer. To do this, it must find a buffer capable of holding the specified number of bytes, allocate a message block and data block, and initialize them so that the message block points to the data block that points to the STREAMS buffer. The caller may then link several messages together to form a segmented message, add the message to a queue, allocate a new message block to form a second reference to some data (for example, in order to retain the data for possible later retransmission), or free up the message.

When sufficient memory is available, *allocb* executes a nearly fixed code sequence[1] that would require 12.5 microseconds in the absence of cache misses. However, measured times ranged from 28 to 198 microseconds, with the average at 64.2 microseconds. We captured a 64.76 microsecond trace on a logic analyzer and found that the worst 19 of the 304 off-chip accesses (6.3%) accounted for 57.6% of the elapsed time and that the worst 31 (10.2%) accounted for 68.4% of the elapsed time.

Behavior of freeb

The *freeb* function typically executes a fixed code sequence that would require 8.8 microseconds in the absence of cache misses. Measured times ranged from 16 to 176 microseconds, with the average at 48.7 microseconds. We captured a 102.8 microsecond trace on a logic analyzer representing a back-to-back pair of *freeb*s invoked from *freemsg*, and found that the worst 28 of the 322 off-chip accesses (8.6%) accounted for 50.6% of the elapsed time, while the worst 74 (23.0%) accounted for 80.3% of the elapsed time.

In both *allocb* and *freeb* the worst accesses were cache misses, either to main memory, to the other processor's cache, or to uncacheable device registers. Note that this behavior is not peculiar to *allocb* or *freeb*; any allocator that consisted of a traditional allocator protected by a simple mutual-exclusion scheme (such as the general-purpose kernel memory allocator) would suffer from the same problem. Other investigators[12] have independently demonstrated some of the difficulties with use of simple mutual exclusion to protect data structures used by traditional algorithms.

An improved version of *allocb* is presented in [6]. This paper describes an improved version of the general-purpose kernel memory allocator.

[1]There is a small loop that selects the proper freelist given the block size, but the maximum execution time for this loop is only a few percent of the total runtime. There are also variations in the number of TLB misses.

Memory Allocator Design

This section presents the design goals that we set out for the new memory allocator, followed by the design itself.

Design Goals

The design goals for the new allocator are:

1) to implement full System V semantics,
2) to support high allocation/deallocation rates,
3) to scale well with increasing processor speeds,
4) to exhibit linear speedup on shared-memory multiprocessors,
5) to be capable of allocating all available buffers to any or all CPUs, and
6) to be capable of coalescing blocks so as to reallocate the memory to different-sized requests.

Implementing full System V semantics adds some overhead. A more efficient interface would allow the caller to request that a given block size be encoded into a "magic cookie" for use in subsequent allocation requests for that size, greatly reducing the number of translations from block size to freelist address. In addition, it is permissible to take the address of the System V allocation (*kmem_alloc*) and deallocation (*kmem_free*) functions. A more efficient interface would also provide C preprocessor macros to perform these functions, thereby avoiding function-call overhead. This paper reports the performance of both the standard version and an optimized version.

An important goal is to exceed the performance of simple global mutual-exclusion. An allocator that meets this goal is faster than any possible ad-hoc allocator based on mutual exclusion; thus, it almost entirely eliminates any motivation to create such ad-hoc allocators. One situation in which ad-hoc allocators are still beneficial is when the structures being allocated are subject to some complex but reusable initialization. The STREAMS buffer allocator described earlier provides an example of this situation. Three different structures (the message block, data block, and data buffer) must be linked together and allocated as a unit. However, the memory allocator's code may be reused for special-purpose allocators such as the STREAMS buffer allocator. This reuse occurs at the binary level,[2] so that a proliferation of special-purpose allocators can be accommodated, if need be, without undue kernel bloat.

A good allocator will scale with the processor speeds as opposed to interconnect latencies. This requires that the allocator exhibit good locality of reference in order to avoid cache-thrashing and that it avoid use of instructions such as read-modify-write

[2]In other words, special-purpose allocators such as *allocb* invoke the same functions as does the general-purpose *kmem_alloc* allocator.

operations and branches that can result in CPU pipeline stalls.

Read-modify-write instructions can result in pipeline stalls because they are required to be executed as if they are atomic. Modern microprocessors operate in a pipelined fashion, overlapping the execution of several instructions. The execution of atomic operations may be overlapped with that of other instructions only under very restricted conditions. Further advances in the art of CPU design might well ease these restrictions. However, superscalar techniques (execution of several parallel pipelines within a single CPU) will increase the penalty associated with stalling for atomic operations.

Branches can result in pipeline stalls because it is not always possible to determine the branch's outcome early enough to do sufficient instruction prefetching. Therefore, the pipeline can stall, waiting for instructions to be fetched from memory or from cache. This effect can be clearly seen in logic-analyzer traces; instruction prefetch will continue along the wrong path when the outcome of a branch is not correctly predicted. The exact magnitude of this effect varies with architecture and with the exact circumstances of the mispredicted branch. However, the amount of effort that has been expended to cause compilers to more accurately predict branches gives some hint of the importance of this effect. Further advances in the arts of compiler and CPU design may make this issue less important, but algorithms such as fully-inlined binary search will likely remain problematic when presented with random input.

Near-linear speedups are needed in order to support configurations with large numbers of processors and communications interfaces. To achieve this goal, the allocator must avoid operations that require coordination between CPUs. An analogy drawn from traffic engineering may be helpful. Within cities, cars must frequently cross each other's paths. Drivers must coordinate their actions (with varying degrees of aggression) in order to avoid collision, and the speed limits are set low to allow for this coordination. In contrast, on rural freeways cars rarely cross each other's paths, and a much lower degree of coordination is required, thereby allowing speed limits to be set higher. Likewise, multiprocessor allocators that avoid the need for coordination avoid inconveniently-low speed limits.

It is clearly important that any given CPU be able to allocate the last remaining buffer, although the allocator is permitted to incur more overhead in this hopefully infrequent low-memory situation.

It is not uncommon for machines in commercial environments to be presented a cyclic workload. For example, the machine might be used for data entry and queries as part of a distributed database during the day, and for backups and database reorganization at night. These different activities often require different sizes of memory allocations, e.g., the data entries and queries might require huge numbers of small blocks of memory to track database locking while the backups and database reorganization might require massive amounts of memory dedicated to user processes.

Consequently, the allocator must be able to coalesce adjacent free blocks of memory into larger blocks, allowing memory to be used to satisfy requests of different sizes or to be returned to the system for use by user processes. Allocators must recover from problems such as overallocation of memory to a given blocksize without a reboot. Coalescing should not interfere with normal system operation, since a one-minute pause caused by an offline coalescing algorithm can be just as disruptive as a reboot.

Roads Not Taken

We considered a number of possible allocation schemes.

Although the McKusick-Karels (MK) algorithm [7] is extremely efficient on uniprocessors when presented with requests whose sizes can be determined at compile-time, it does not meet goals 3 and 4 on multiprocessors. In particular, its fully-inlined binary search results in pipeline stalls because no reasonable instruction prefetch strategy can correctly predict all of the branches. As presented, the MK algorithm also fails to meet goal 6, but could be modified to do the required coalescing. Nonetheless, the large number of algorithms that are directly and indirectly derived from the MK algorithm (including the one presented in this paper) form an impressive testament to its strengths.

One such algorithm is the watermark-based lazy buddy system[5], which attempts to combine high-speed allocation with high-quality coalescing. However, it requires global synchronization on each operation and fails to maintain good locality of reference (since each block is sent singly to be coalesced, rather than being sent in large groups), thereby failing to meet goals 3 and 4 on multiprocessors.

Another MK-derived algorithm is Rogue Wave's C++ memory allocator [8]. This allocator also attempts to combine high-speed allocation with high-quality coalescing, but intentionally degrades its ability to coalesce in favor of decreasing the resident set size of the program. This is a laudable goal within a user process, but is largely irrelevant within the kernel. Furthermore, the algorithm is not designed for use on multiprocessors and so does not meet goals 3 and 4 in this environment.

Algorithms designed specifically to promote high-quality coalescing [3] are quite slow [4] and thus fail to meet goal 2. It is quite difficult to exceed the performance of removing the first element from a simple, singly-linked, linear list.

Allocator Design

The requirements for high speed and for coalescing conflict to a large degree. Very little coalescing can be performed within the 9-VAX-instruction budget of the McKusick-Karels allocator. It is nevertheless possible to do both high-speed allocation and high-quality, online coalescing by introducing the concept of layering to the allocator.

The allocator consists of four layers:

1) a per-CPU caching layer,
2) a global layer,
3) a coalesce-to-page layer, and
4) a coalesce-to-"vmblk" layer.

The lower-numbered layers are optimized for speed, while the the higher-numbered layers are optimized for coalescing, as illustrated in Figure 1.

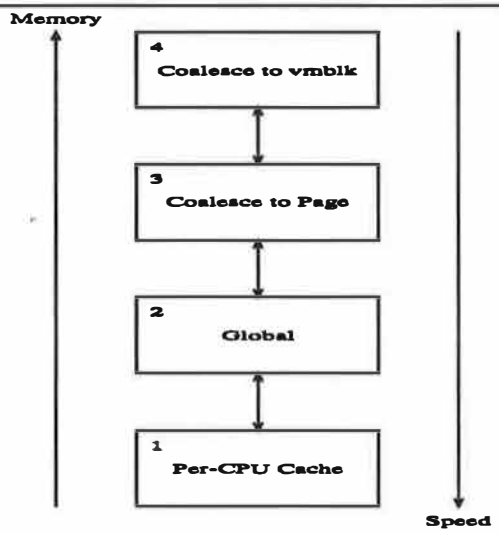


Figure 1: Allocator Layering

The following sections describe each of these layers in turn. A final section describes how "cookies" are used to efficiently encapsulate request-size information.

Per-CPU Caching Layer

The only purpose of the per-CPU caching layer is to support high-speed allocation and deallocation in the common case. Each CPU maintains a local cache of buffers for each of a small fixed set of buffer sizes, much as the McKusick-Karels algorithm does. Consequently, there is one instance of a per-CPU cache for each possible CPU-buffer-size combination. For example, a four-CPU system that managed the default set of nine power-of-two block sizes (16, 32, 64, 128, 254, 512, 1024, 2048, and 4096 bytes) would have 36 per-CPU caches.

The *kmem_alloc* function first attempts to satisfy a request for a given size of block from the appropriate cache on the current CPU. For example, an interrupt routine running on CPU 2 needing a 50-byte block would first check CPU 2's cache of 64-byte blocks. CPUs are prohibited from accessing other CPUs' per-CPU caches, thus removing the need for any synchronization primitives (other than the disabling of interrupts) guarding the per-CPU caches.

When a per-CPU cache is exhausted, it is replenished from the global layer; when it becomes too full (as determined by a kernel parameter named *target*), the excess is put back into the global layer. Blocks are moved in *target*-sized groups, preventing unnecessary linked-list operations. This is accomplished by maintaining a split freelist in the per-CPU cache as shown in Figure 2.

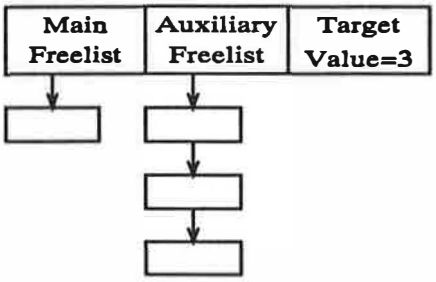


Figure 2: Per-CPU Data Structures

The maximum size of each half of the per-CPU freelist is *target*, so that the total number of blocks in a per-CPU freelist may range up to twice *target*. Blocks are normally allocated from and freed to the *main* list. If adding another block would cause the *main* list to exceed *target*, *main* is moved to *aux*. If *aux* is not empty, its contents are first returned to the global layer. Thus, as shown in Figure 2, up to two additional blocks may be freed onto *main*. Freeing a third block would cause the contents of *aux* to be returned to the global pool, the contents of *main* to be moved to *aux*, and the newly-freed block to be added to *main*. At this point, the configuration would again be as shown in Figure 2.

If *main* is empty upon allocation, the contents of *aux*, if any, are moved to *main*. If *aux* is also empty, *main* is instead replenished from the global layer. In the situation shown in Figure 2, one more block may be allocated from *main*, at which point *main* will be empty. A second allocation will result in the contents of *aux* being moved to *main* and one of the blocks being used to satisfy the allocation request. At this point, *main* will contain two more blocks and *aux* will be empty, allowing two additional allocations to be made directly from *main*. The next allocation would find both *main* and *aux* empty, causing *main* to be refilled from the global layer.

Note that the global layer will be accessed at most one time per *target*-number of accesses. This means that the per-allocation overhead incurred in the global layer may be reduced to any desired level simply by increasing the value of *target*. The only penalty for increasing *target* is the increased amount

of memory that will reside in the per-CPU caches. In practice, there is no motivation to increase *target* beyond the point at which the global-layer overhead becomes an insignificant portion of the per-allocation overhead.

Global Layer

The only purpose of the global layer is to support reasonable performance in cases when one CPU allocates buffers of a given size, which are then passed to other CPUs that free them. The global layer allows the freed buffers to move back to the allocating CPU without incurring the overhead of coalescing.

There is a separate instance of the global layer for each block size. Each instance maintains free blocks in lists of *target*-sized lists, as shown in Figure 3.

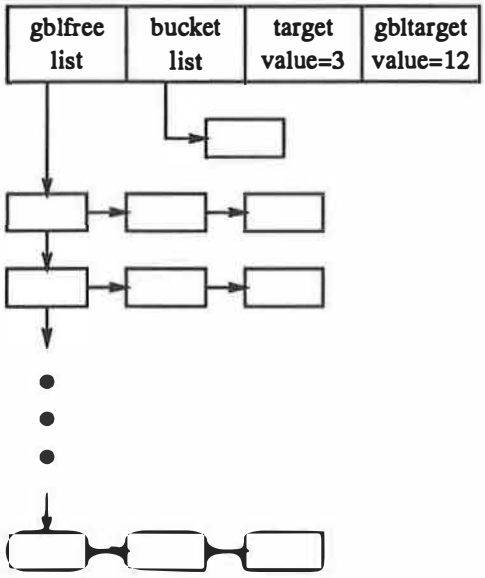


Figure 3: Global Layer Data Structures

This technique allows *target*-sized blocks of data to be passed to and from the per-CPU layers with a minimum number of linked-list operations. Odd-sized lists of blocks may be passed into the global layer during low-memory operation or during per-CPU cache flushes. These lists are added to the *bucket* list, which is used to group the blocks back into *target*-sized lists.

When the global layer becomes too full, the excess buffers are sent up to the coalesce-to-page layer. When the global layer is empty, it is replenished from the coalesce-to-page layer. The number of blocks in the global layer ranges up to twice a parameter named *gbltarget*. There is no reason to maintain a split freelist at the global layer, since each block must be individually examined by the coalesce-to-page layer (described in the following section) in order to determine which page's freelist it belongs on.

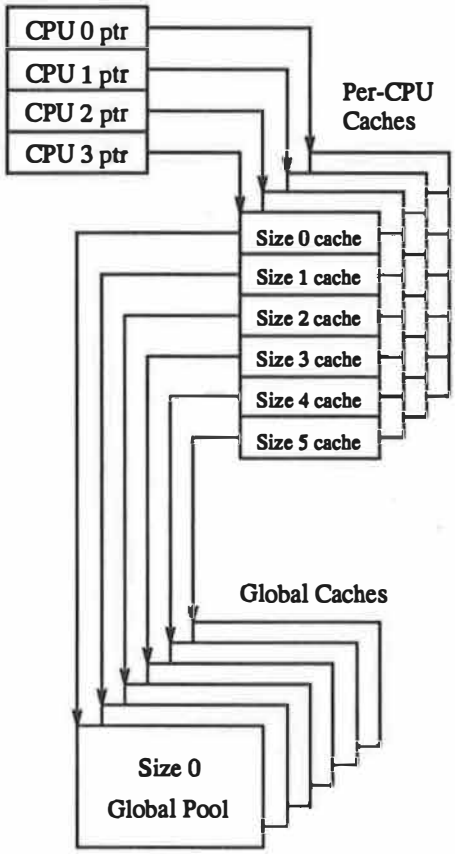


Figure 4: Per-CPU and Global Layers

A schematic view of the data structures implementing the per-CPU and global layers is shown in Figure 4. Each CPU has a pointer to an array of its per-CPU caches, and each per-CPU cache maintains a pointer to the global pool serving its blocksize. Request sizes are converted to indexes into the array of caches through use of a table indexed by size.

Coalesce-to-Page Layer

The coalesce-to-page layer gathers blocks of a given size and coalesces them into pages. This layer maintains an auxiliary data structure for each page, which contains the per-page freelist and a count of the number of blocks in the page that are currently free (this per-page data structure is described in more detail in the discussion of the coalesce-to-vmblk layer below). When the count equals the total number of blocks in the page, the entire page may be given back to the system; in other words, the coalesce-to-page layer can immediately determine when all of the blocks in a given page have been freed up. This eliminates the need for a computationally-expensive mark-and-sweep algorithm or an offline sorting algorithm. Pages that have some blocks in use are placed on a radix-sorted freelist so that pages with the fewest free blocks will be allocated from most frequently, as shown in Figure 5.

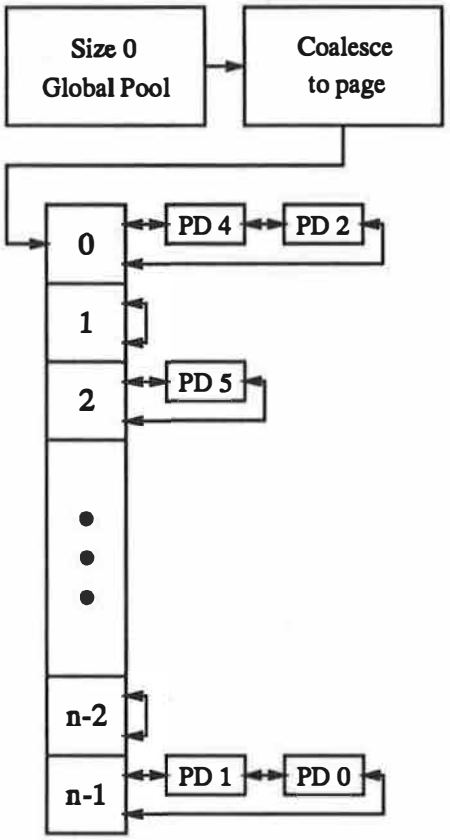


Figure 5: Coalesce-to-Page Layer

This sorting has the benefit of allowing pages that have only a few in-use blocks more time to gather them. In turn, this allows the page to be used for allocations of other sizes and for user processes.

Once all of the blocks in a page have been freed, the physical memory is returned to the system. The virtual memory is retained and passed up to the coalesce-to-vmblk layer. This process illustrates a key difference between kernel- and user-level memory allocators. Kernel-level allocators must manage the virtual address space and physical memory explicitly and separately. In contrast, user-level allocators need not and typically cannot easily distinguish between virtual and physical memory.

Coalesce-to-vmblk Layer

This layer manages large vmblks of virtual memory (4 megabytes in size for the current implementation). Pages of virtual-address space are allocated from vmblks as needed and are mapped onto physical memory. Requests for blocks of memory larger than one page bypass layers 1 through 3 and are handled directly by the coalesce-to-vmblk layer. Adjacent spans of free pages in a vmblk are coalesced as they are freed; a boundary-tag-like scheme uses per-page auxiliary data structures (called page descriptors) to track the sizes and locations of free spans of virtual memory.

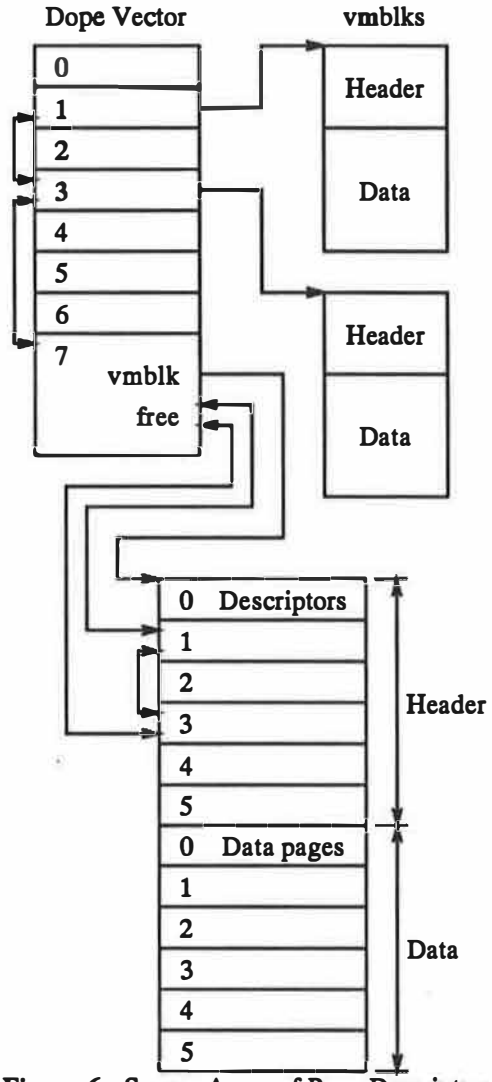


Figure 6: Sparse Array of Page Descriptors

The system must be able to locate the page descriptor corresponding to a particular block given only that block's address. This is accomplished with a two-level scheme using a sparse array as shown in Figure 6. In the first level the upper bits of the block's address are used to index into a dope vector, which contains the address of the vmblk containing that block. The vmblk consists of a group of page descriptors followed by the corresponding data pages. In the second level, the index of the block's page descriptor within the vmblk is obtained by subtracting the vmblk's address from the block's address, shifting off the lower bits to get the page index within the vmblk, and finally subtracting the number of pages occupied by the page descriptors.

This two-level scheme allows overhead information to be kept only for those pages controlled by the allocator. Other pages (such as those used by processes) require no such overhead. The performance penalty associated with this two-level scheme is incurred only at the coalesce-to-page and

coalesce-to-vmblk layers, and therefore has a minimal effect on overall system performance.

Page descriptors corresponding to pages that have been split into blocks contain the block size, a freelist pointer, and the number of free blocks. Page descriptors corresponding to spans contain the boundary-tag information and free-list pointers needed to allocate and coalesce large blocks.

Cookies

As noted earlier, there is significant overhead associated with inlined binary searches given widely-varying inputs that defeat branch-prediction schemes. Hence, the inline binary search used by the MK algorithm is most effective when the size is known at compile time. Otherwise, a subroutine call combined with a table lookup can be just as efficient.

Explicitly requiring that the request size be known at compile time allows the overhead of freeing to be further reduced (cases where the request size is not known at compile time may be handled by the standard function interface). The caller invokes *kmem_alloc_get_cookie* to translate a request size into an opaque "cookie" that is passed to subsequent expansions of the macros named *KMEM_ALLOC_COOKIE* and *KMEM_FREE_COOKIE*. The cookie contains pointers to the proper per-CPU pools, removing the need for the free operation to determine the block size given only its address.

The use of cookies allows the common case of the free operation to consume only thirteen 80x86 instructions, as compared to the 16 VAX instructions consumed by the MK algorithm.

Measurements

The following sections present instruction counts for the allocator, measurements on a simple benchmark that exhibits best-case performance, measurements on another simple benchmark that exhibits worst-case performance, and finally measurements taken from a more sophisticated benchmark that makes more typical use of the allocator.

All measurements were taken on a Symmetry 2000 system with 50MHz 80486 processors.

Instruction Counts

The efficient "cookie" version of the allocator executes thirteen 80x86 instructions each for the allocation and free operations. Allocation overhead is comparable to that of MK when differences between the VAX and 80x86 instruction set are taken into account (in particular, the 80x86 lacks a memory-to-memory move instruction). A single additional memory reference is required in order to handle multiple processors. The overhead of freeing is somewhat less than that of MK even without considering instruction-set differences. The difference is due to the use of the cookie-based scheme. MK must look up the block's size and use this information to index into the list of freelist, while the cookie allows direct access to the proper per-CPU cache.

Note that the efficient version is nonstandard and is useful only when the size of the request is known at compile-time.

The less efficient but standard interface executes 35 instructions for allocation and 32 instructions for freeing, assuming that the each of the actual arguments can be evaluated and stored with a single instruction. The additional overhead is caused by the function call and by the need to map from the request size to the proper per-CPU cache. Currently, all variable-sized structures have large initialization overheads that overwhelm the performance difference between the standard and cookie-based interfaces.[3] Therefore, there is currently little motivation to provide a third interface that provides speedier allocation of variable-length structures.

Best-Case Benchmark

We measured best-case performance by constructing a system call containing a loop that is run for a user-specified length of time. Each pass through the loop invokes *kmem_alloc* to allocate a buffer, then invokes *kmem_free* to immediately deallocate this same buffer. When the specified length of time has passed, the system call returns the number of *kmem_alloc*/*kmem_free* pairs that were executed. Thus, the measurements include the overhead of the loop which invokes *kmem_alloc* and *kmem_free* ; this overhead amounts to as much as 40% for the faster algorithms. This system call is invoked from a user program, which is forced to run on a specified CPU. Multiple-CPU data is collected by running multiple instances of the program, each on its own CPU.

The performance was highly linear as shown in Figure 7. The x-axis shows the number of CPUs and the y-axis shows the number of pairs of allocation and freeing accomplished per second. The top trace shows the performance of the non-standard cookie-based macro, the next trace shows the performance of the standard functional interface, and the bottom two traces show the performance of naive parallelizations of the MK algorithm and of the "oldkma" algorithm, which resembles "Fast Fits" [11] (algorithm "S" in Korn's and Vo's survey [4]).

Figure 8 displays the same data on a semilog plot so that the traces for the two slower algorithms may be more easily distinguished from the x-axis. The irregularities in the trace of the naive parallelization of the MK algorithm are due to second-order effects resulting from the extreme lock contention exhibited by this algorithm. These effects are

[3]The only exception to this rule is the communications subsystem, for which a special-purpose allocator (*allocb* and *freeb*) already exists.

largely masked by the greater overhead of the slower "oldkma" algorithm.

The cookie-based allocator ranges from 15 times the performance of the "oldkma" allocator on a single CPU to more than 1,000 times the performance on 25 CPUs.[4] The standard interface is roughly half as fast as the cookie-based allocator, but note that this dramatic-seeming difference in performance amounts to only about 20 instructions per operation.

In contrast, the other two schemes simply did not scale with increasing numbers of CPUs. In fact, in both cases, the best performance was observed when running on a single CPU.

Hardware monitors indicate that the common case of the two fast algorithms are free from the cache-thrashing that accounted for so much of the original algorithm's execution time. We therefore expect that the allocator will continue to scale well with increasing processor speeds.

[4]Although the machine we were using had 26 CPUs, we cannot reliably measure the performance of all 26 CPUs simultaneously because the script that coordinates the tests must use one of the CPUs.

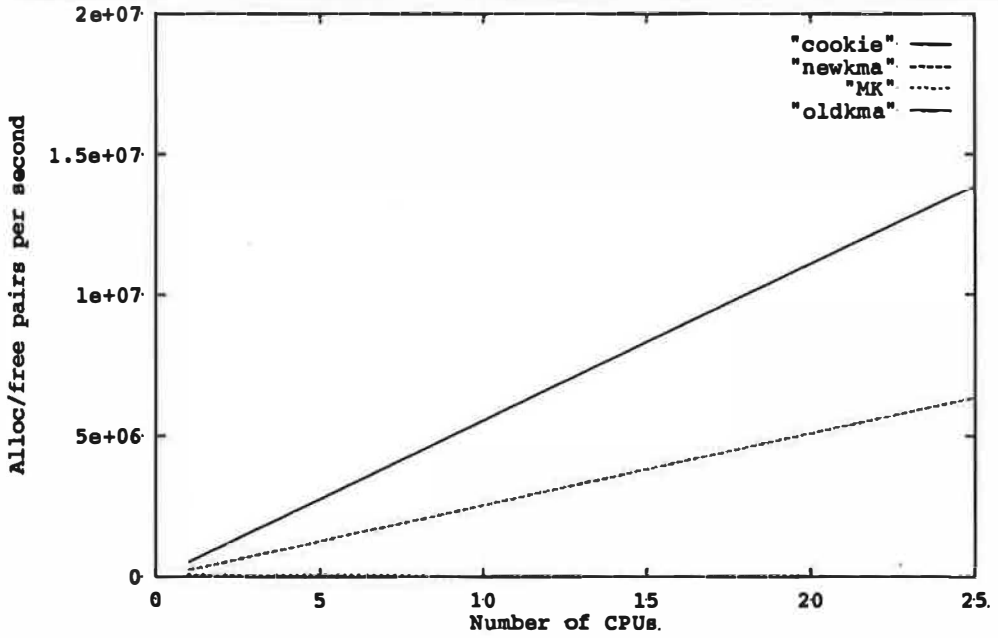


Figure 7: Performance of New *kmem_alloc* and *kmem_free*

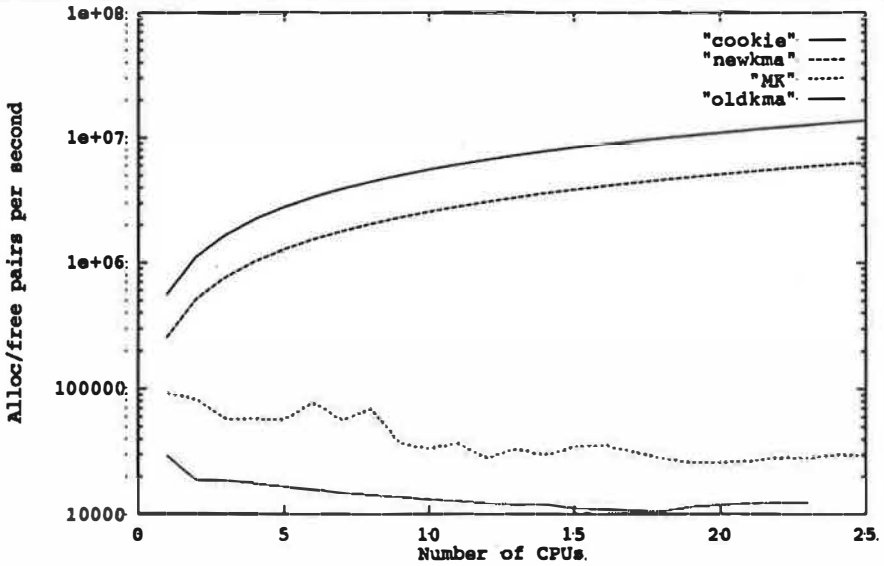


Figure 8: Performance of New *kmem_alloc* and *kmem_free*

Worst-Case Benchmark

The best-case benchmark exercises only the per-CPU caching layer. The worst-case benchmark exercises not only all the layers, but takes care to exercise the upper layers to the greatest extent possible, thereby incurring the worst possible per-allocation overhead. This is accomplished by allocating blocks of a given size until memory is exhausted, freeing them all, then repeating the process with the next-larger size.

The benchmark is implemented as a shell script which uses a set of special-purpose system calls which allow the user to explicitly specify sequences of allocation and free operations. A *syscall_kma*() system call causes the system to allocate a specified number of blocks of a given size, placing them on a linked list in the kernel. A companion *syscall_kmf*() system call causes the system to free a specified number of blocks from the linked list.

Note that an allocator that does no coalescing would fail to complete this benchmark, having permanently fragmented all available memory into the smallest possible blocks. It would be necessary to reboot the system between runs of each block size. An allocator that does periodic offline coalescing would require that appropriate *sleep* commands be placed in the script in order to ensure that the newly-freed blocks of the previous size were fully coalesced before advancing to the next size. The fact that our allocator required neither reboots nor delays of any sort demonstrates the effectiveness of the coalescing scheme.

The results are shown on Figure 9. Note that the x-axis is in units of block size rather than number of CPUs. Large blocks showed decreased performance because they require physical memory to be allocated from the virtual-memory system more frequently, and the *target* value is set by a heuristic that limits the amount of memory that is tied up in per-CPU caches. This value ranges from 10 for 16-byte blocks to just 2 for 4096-byte blocks. Although this heuristic may be overridden to increase performance, there is usually little reason to. The overhead of initializing large blocks of memory typically overshadows the virtual-memory system's overhead.

Freeing small blocks is more expensive than allocating them because of the overhead of mapping from the block's address to its per-page freelist. Normally, this overhead would be infrequently incurred, but the worst-case benchmark forces it to occur on each and every free.

Distributed Lock Manager Benchmark

The best-case benchmark is effectively measuring only the performance of the per-CPU layer, while the worst-case benchmark overstates the overhead of the upper layers. Realistically evaluating the overall performance requires measuring an application that makes more sophisticated use of the memory allocator than did the simple benchmarks presented in the previous sections. The application we selected was a distributed lock manager, which makes heavy use of *kmem_alloc* in order to build data structures needed to track lock requests and ownership. This lock manager is used by OLTP applications to maintain a consistent view of data among a cooperating cluster of machines.

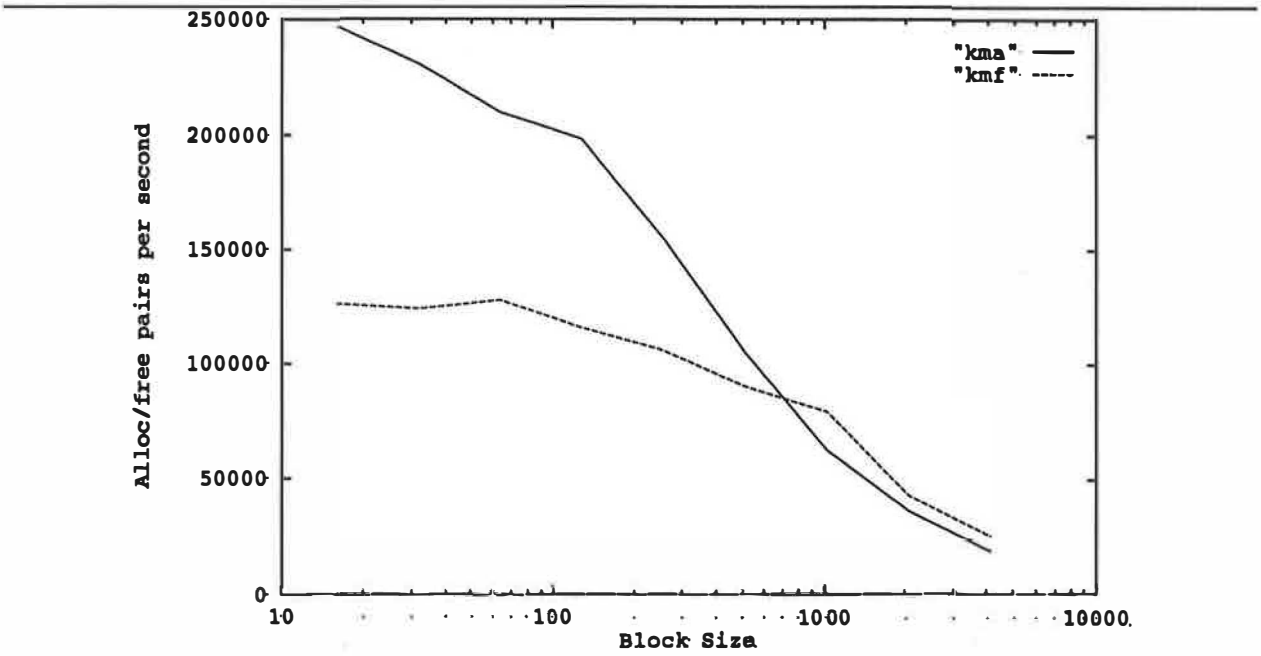


Figure 9: Worst-Case Performance

Unfortunately, it is not possible to directly measure the *kmem_alloc* overhead in this benchmark. The microsecond counters used to measure the overhead for the two simple benchmarks do not have enough resolution to accurately measure an isolated invocation of these allocators. However, the degree by which the upper layers will degrade performance can be expressed in terms of miss rates. We define the miss rate at a given layer as the fraction of accesses to that layer that require the services of a higher layer. For the value of 10 used for *target* for small blocks, at most one of every ten allocations will require the services of the global layer. Hence, the maximum miss rate from the per-CPU caching layer is 10%. The value of 15 used for *gbltarget* for small blocks results in a maximum miss rate of 6.7% from the global layer to the coalescing layer. The maximum combined miss rate from the per-CPU and global layers is 0.67%. In other words, at most one out of every 150 allocations will require service from the coalescing layer. Real applications will fall somewhere between the best- and worst-case benchmarks. Measuring a particular application's miss rates allows us to estimate that application's allocation overhead without the need for special-purpose hardware.

The miss rate from the per-CPU layer into the global layer ranged from 2.1% (for frees of 256-byte blocks) to 7.8% (for allocations of 512-byte blocks). Note that the 7.8% figure is fairly close to the worst-case figure of 10%. Again, if need be, the value of *target* can be increased to reduce both the worst-case and the real-world miss rates.

The miss rate from the global layer to the coalesce-to-page layer ranged from 1.2% (for frees of 256-byte blocks) to 3.0% (for allocations of 512-byte blocks). Both these figures compare favorably to the worst-case figure of 6.7%.

The combined miss rate of the per-CPU and global layers to the coalesce-to-page layer ranged from 0.02% (frees of 256-byte blocks) to 0.14% (allocations of 512-byte blocks), both of which compare favorably to the worst case of 0.67%. These combined miss rates ensure that coalescing overhead is diluted by a factor ranging from 700 to 5000, thus maintaining an acceptable per-block overhead.

Conclusions

The new *kmem_alloc* and *kmem_free* functions meet their design goals. These goals are achieved by avoiding synchronization, by taking advantage of cache locality (rather than through use of sophisticated synchronization schemes), and by maintaining low miss rates at the per-CPU and global layers so as to dilute the overhead inherent in coalescing.

These functions are more than capable of meeting the challenge of commercial data processing. They also clearly demonstrate that the problem of efficient resource allocation on a shared-memory multiprocessor is quite tractable.

Acknowledgments

We are grateful to Bob Miller, Steve Neuner, and Dilip Ratnam, who exhibited unbelievable intestinal fortitude in allowing their tight development schedules to depend on our unproven (and, at the time, uncompleted) implementation.

We owe much to Robin O'Neill for patiently answering our questions about the kernel implementation (even the stupid ones), and to Corene Casper for doggedly tracking down a number of problems that turned out to be bugs in early versions of our code. Brent Kingsbury's thorough and thoughtful review of our design saved us much time and effort, and Gary Graunke's insight into parallel algorithms provided invaluable guidance. Mick O'Halloran, Elizabeth Strohecker, and David Wolfe provided the hardware needed to run our performance tests.

We owe many thanks to Jeff Berkowitz and the anonymous referees for their careful technical review of the paper, and to James Bash for helping to render this paper human-readable.

Finally, we are greatly indebted to Steve Neuner and Jon Simms for their unwavering support of this effort.

References

[1] Maurice Herlihy. Wait-free synchronization. *ACM TOPLAS*, 11(1):124-149, January 1991.

[2] John L. Hennessy and Norman P. Jouppi. Computer technology and architecture: An evolving interaction. *IEEE Computer*, pages 18-28, September 1991.

[3] Donald Knuth. *The Art of Computer Programming*. Addison-Wesley, 1973.

[4] David G. Korn and Kiem-Phong Vo. In search of a better malloc. In *USENIX Conference Proceedings*, Berkeley CA, June 1985.

[5] T. Paul Lee and R. E. Barkley. Design and evaluation of a watermark-based lazy buddy system. *Performance Evaluation Review*, 17(1), May 1989.

[6] Paul E. McKenney and Gary Graunke. Efficient buffer allocation on shared-memory multiprocessors. In *IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, Tucson, AZ, February 1992.

[7] Marshall Kirk McKusick and Michael J. Karels. Design of a general purpose memory allocator for the 4.3BSDUNIXkernel. In *USENIX Conference Proceedings*, Berkeley CA, June 1988.

[8] Nathan Myers. C++ memory management: An overview. Message-ID 9210131855.-

AA24066@rwave.roguewave.com, October 1992.

[9] D. M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, October 1984.

[10] Harold S. Stone and John Cocke. Computer architecture in the 1990s. *IEEE Computer*, pages 30-38, September 1991.

[11] C. J. Stephenson. Fast fits: New methods for dynamic storage allocation. *SIGOPS Operating System Review*, 17(5), 1983.

[12] Josep Torrellas, Anoop Gupta, and John Hennessy. Characterizing the caching and synchronization performance of a multiprocessor operating system. In *ASPLOS V*, October 1992.

Author Information

Paul E. McKenney received BS-CS and BS-ME degrees from Oregon State University in 1981. He worked as a contract programmer for four years, then joined SRI International first as a system administrator and later as a research engineer in packet-radio and wide-area-network communications. He is currently working as a performance analyst for Sequent Computer Systems.

Jack Slingwine graduated from Pennsylvania State University in 1978 with an Associate's Degree in Computer Science. He has recently been involved in the design and implementation of UNIX System V ES/MP and is currently a member of Sequent's Clustered Systems Group where he is involved in all aspects of Unix operating-system design and implementation.

Reach them at 15450 SW Koll Parkway Beaverton, OR 97006 (503) 578-5131, via e-mail at {mckenney,jacks}@sequent.com, or FAX (503) 578-5271.

An Implementation of a Log-Structured File System for UNIX

Margo Seltzer -- Harvard University
Keith Bostic -- University of California, Berkeley
Marshall Kirk McKusick -- University of California, Berkeley
Carl Staelin -- Hewlett-Packard Laboratories

ABSTRACT

Research results [ROSE91] suggest that a log-structured file system (LFS) offers the potential for dramatically improved write performance, faster recovery time, and faster file creation and deletion than traditional UNIX file systems. This paper presents a redesign and implementation of the Sprite [ROSE91] log-structured file system that is more robust and integrated into the vnode interface [KLEI86]. Measurements show its performance to be superior to the 4BSD Fast File System (FFS) in a variety of benchmarks and not significantly less than FFS in any test. Unfortunately, an enhanced version of FFS (with read and write clustering) [MCVO91] provides comparable and sometimes superior performance to our LFS. However, LFS can be extended to provide additional functionality such as embedded transactions and versioning, not easily implemented in traditional file systems.

1. Introduction

Early UNIX file systems used a small, fixed block size and made no attempt to optimize block placement [THOM78]. They assigned disk addresses to new blocks as they were created (preallocation) and wrote modified blocks back to their original disk addresses (overwrite). In these file systems, the disk became fragmented over time so that new files tended to be allocated randomly across the disk, requiring a disk seek per file system read or write even when the file was being read sequentially.

The Fast File System (FFS) [MCKU84] dramatically increased file system performance. It increased the block size, improving bandwidth. It reduced the number and length of seeks by placing related information close together on the disk. For example, blocks within files were allocated on the same or a nearby cylinder. Finally, it incorporated rotational disk positioning to reduce delays between accessing sequential blocks.

The factors limiting FFS performance are synchronous file creation and deletion and seek times between I/O requests for different files. The synchronous I/O for file creation and deletion provides file system disk data structure recoverability after failures. However, there exist alternative solutions such as NVRAM hardware [MORA90] and logging software [KAZA90]. In a UNIX environment, where the vast majority of files are small [OUST85] [BAKE91], the seek times between I/O requests for different files can dominate. No solutions to this problem currently exist in the context of FFS.

The log-structured file system, as proposed in [OUST89], attempts to address both of these problems. The fundamental idea of LFS is to improve file system performance by storing all file system data in a single, continuous log. Such a file system is optimized for writing, because no seek is required between writes. It is also optimized for reading files written in their entirety over a brief period of time (as is the norm in UNIX systems), because the files are placed contiguously on disk. Finally, it provides temporal locality, in that it is optimized for accessing files that were created or modified at approximately the same time.

The write-optimization of LFS has the potential for dramatically improving system throughput, as large main-memory file caches effectively cache reads, but do little to improve write performance [OUST89]. The goal of the Sprite log-structured file system (Sprite-LFS) [ROSE91] was to design and implement an LFS that would provide acceptable read performance as well as improved write performance. Our goal is to build on the Sprite-LFS work, implementing a new version of LFS that provides the same recoverability guarantees as FFS, provides performance comparable to or better than FFS, and is well-integrated into a production quality UNIX system.

This paper describes the design of log-structured file systems in general and our implementation in particular, concentrating on those parts that differ from the Sprite-LFS implementation. We compare the performance of our implementation of LFS (BSD-LFS) with FFS using a variety of benchmarks.

2. Log-Structured File Systems

There are two fundamental differences between an LFS and a traditional UNIX file system, as represented by FFS; the on-disk layout of the data structures and the recovery model. In this section we describe the key structural elements of an LFS, contrasting the data structures and recovery to FFS. The complete design and implementation of Sprite-LFS can be found in [ROSE92]. Table 1 compares key differences between FFS and LFS. The reasons for these differences will be described in detail in the following sections.

2.1. Disk Layout

In both FFS and LFS, a file's physical disk layout is described by an index structure (*inode*) that contains the disk addresses of some *direct*, *indirect*, *doubly indirect*, and *triply indirect blocks*. Direct blocks contain data, while indirect blocks contain disk addresses of direct blocks, doubly indirect blocks contain disk addresses of indirect blocks, and triply indirect blocks contain disk addresses of doubly indirect blocks. The inodes and single, double and triple indirect blocks are referred to as "meta-data" in this paper.

The FFS is described by a *superblock* that contains file system parameters (block size, fragment size, and file system size) and disk parameters (rotational delay, number of sectors per track, and number of cylinders). The superblock is replicated throughout the file system to allow recovery from crashes that corrupt the primary copy of the superblock. The disk is statically partitioned into cylinder groups, typically between 16 and 32 cylinders to a group. Each group contains a fixed number of inodes (usually one inode for every two kilobytes in the group) and bit maps to record inodes and data blocks available for allocation. The inodes in a cylinder group reside at fixed disk addresses, so that disk addresses may be computed from inode numbers. New blocks are allocated to optimize for sequential file access. Ideally, logically sequential blocks of a file are allocated so that no seek is required between two consecutive accesses. Because data blocks for a file are typically accessed together, the FFS policy routines try to place data blocks for a file in the same cylinder group, preferably at rotationally optimal positions in the same cylinder. Figure 1 depicts the physical layout of FFS.

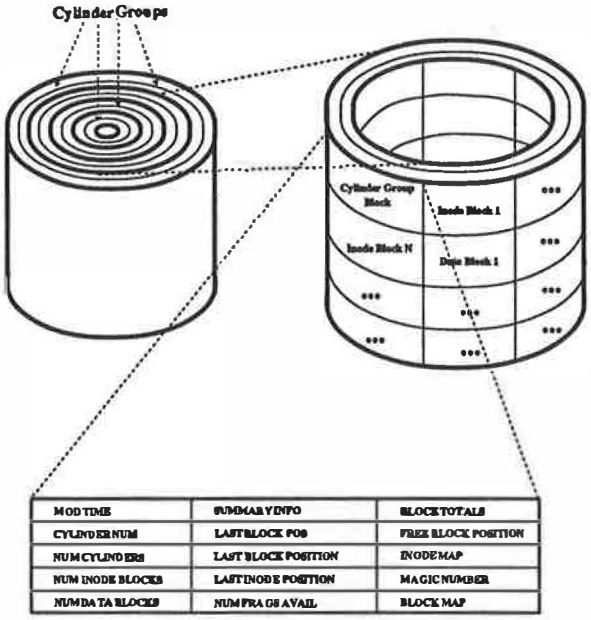


Figure 1: Physical Disk Layout of the Fast File System. The disk is statically partitioned into cylinder groups, each of which is described by a cylinder group block, analogous to a file system superblock. Each cylinder group contains a copy of the superblock and allocation information for the inodes and blocks within that group.

Task	FFS	LFS
Assign disk addresses	block creation	segment write
Allocate inodes	fixed locations	appended to log
Maximum number of inodes	statically determined	grows dynamically
Map inode numbers to disk addresses	static address	lookup in inode map
Maintain free space	bit maps	cleaner segment usage table
Make file system state consistent	fsck	roll-forward
Verify directory structure	fsck	background checker

Table 1: Comparison of File System Characteristics of FFS and LFS.

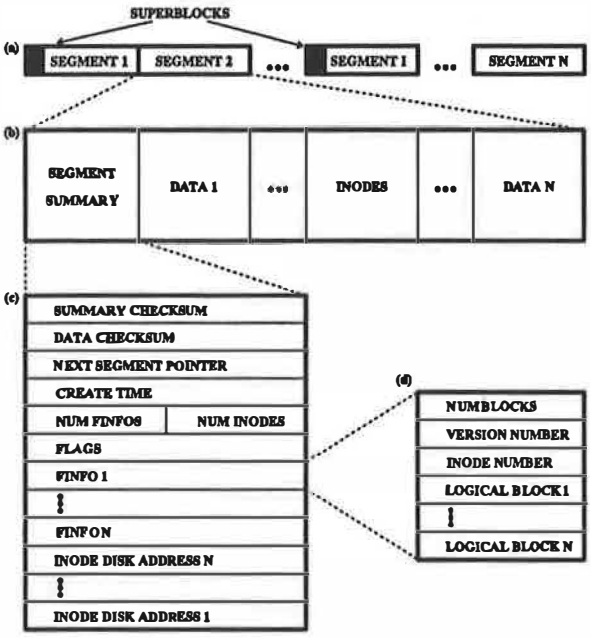


Figure 2: A Log-Structured File System. A file system is composed of segments as shown in Figure (a). Each segment consists of a summary block followed by data blocks and inode blocks (b). The segment summary contains checksums to validate both the segment summary and the data blocks, a timestamp, a pointer to the next segment, and information that describes each file and inode that appears in the segment (c). Files are described by FINFO structures that identify the inode number and version of the file (as well as each block of that file) located in the segment (d).

LFS is a hybrid between a sequential database log and FFS. It performs all writes sequentially, like a database log, but incorporates the FFS index structures into this log to support efficient random retrieval. In an LFS, the disk is statically partitioned into fixed size segments, typically one-half megabyte. The logical ordering of these segments creates a single, continuous log.

An LFS is described by a superblock similar to the one used by FFS. When writing, LFS gathers many dirty pages and prepares to write them to disk sequentially in the next available segment. At this time, LFS sorts the blocks by logical block number, assigns them disk addresses, and updates the meta-data to reflect their addresses. The updated meta-data blocks are gathered with the data blocks, and all are written to a segment. As a result, the inodes are no longer in fixed locations, so, LFS requires an additional data structure, called the *inode map* [ROSE90], that maps inode numbers to disk addresses.

Since LFS writes dirty data blocks into the next available segment, modified blocks are written to the disk in different locations than the original blocks. This space reallocation is called a "no-overwrite" policy, and it necessitates a mechanism to reclaim space resulting from deleted or overwritten blocks. The *cleaner* is a garbage collection process that reclaims space from the file system by reading a segment, discarding "dead" blocks (blocks that belong to deleted files or that have been superseded by newer blocks), and appending any "live" blocks. For the cleaner to determine which blocks in a segment are "live," it must be able to identify each block in a segment. This determination is done by including a summary block in each segment that identifies the inode and logical block number of every block in the segment. In addition, the kernel maintains a *segment usage table* that shows the number of "live" bytes and the last modified time of each segment. The cleaner uses this table to determine which segments to clean [ROSE90]. Figure 2 shows the physical layout of the LFS.

While FFS flushes individual blocks and files on demand, the LFS must gather data into segments. Usually, there will not be enough dirty blocks to fill a complete segment [BAKE92], in which case LFS writes *partial segments*. A physical segment contains one or more partial segments. For the remainder of this paper, *segment* will be used to refer to the physical partitioning of the disk, and *partial segment* will be used to refer to a unit of writing. Small partial segments most commonly result from NFS operations or *fsync*(2) requests, while writes resulting from the *sync*(2) system call or system memory shortages typically form larger partials, ideally taking up an entire segment. During a *sync*, the inode map and segment usage table are also written to disk, creating a *checkpoint* that provides a stable point from which the file system can be recovered in case of system failure. Figure 3 shows the details of allocating three files in an LFS.

2.2. File System Recovery

There are two aspects to file system recovery: bringing the file system to a physically consistent state and verifying the logical structure of the file system. When FFS or LFS add a block to a file, there are several different pieces of information that may be modified: the block itself, the inode, the free block map, possibly indirect blocks, and the location of the last allocation. If the system crashes during the addition, the file system is likely be left in a physically inconsistent state. Furthermore, there is currently no way for FFS to localize inconsistencies. As a result, FFS must rebuild the entire file system state, including cylinder group bit maps and meta-data. At the same

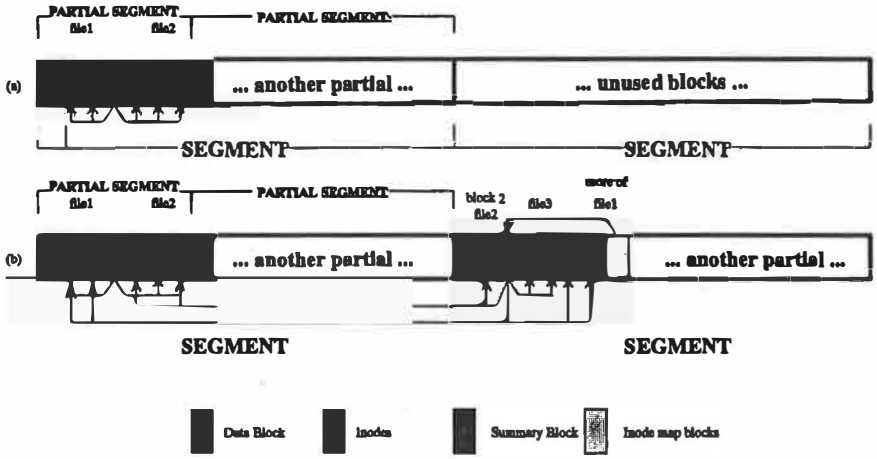


Figure 3: File Allocation in a Log-Structured File System. In figure (a), two files have been written, file1 and file2. Each has an index structure in the meta-data block that is allocated after it on disk. In figure (b), the middle block of file2 has been modified. A new version of it is added to the log, as well as a new version of its meta-data. Then file3 is created, causing its blocks and meta-data to be appended to the log. Next, file1 has two more blocks appended to it, causing the blocks and a new version of file1's meta-data to be appended to the log. On checkpoint, the *inode map* containing pointers to the meta-data blocks, is written.

time, FFS verifies the directory structure and all block pointers within the file system. Traditionally, *fsck*(8) is the agent that performs both of these functions.

In contrast to FFS, LFS writes only to the end of the log and is able to locate potential inconsistencies and recover to a consistent physical state quickly. This part of recovery in LFS is more similar to standard database recovery [HAER83] than to *fsck*. It consists of two parts: initializing all the file system structures from the most recent checkpoint and then "rolling forward" to incorporate any modifications that occurred subsequently. The roll forward phase consists of reading each segment after the checkpoint in time order and updating the file system state to reflect the contents of the segment. The next segment pointers in the segment summary facilitate reading from the last checkpoint to the end of the log, the checksums are used to identify valid segments, and the timestamps are used to distinguish the partial segments written after the checkpoint and those written before which have been reclaimed. The file and block numbers in the FINFO structures are used to update

Phase I	Traverse inodes Validate all block pointers. Record inode state (allocated or unallocated) and file type for each inode. Record inode numbers and block addresses of all directories.
Phase II	Sort directories by disk address order. Traverse directories in disk address order. Validate ".". Record "..". Validate directories' contents, type, and link counts. Recursively verify "..".
Phase III	Attach any unresolved ".." trees to lost+found. Mark all inodes in those trees as "found".
Phase IV	Put any inodes that are not "found" in lost+found. Verify link counts for every file.
Phase V	Update bit maps in cylinder groups.

Table 2: Five Phases of *fsck*.

the inode map, segment usage table, and inodes making the blocks in the partial segment extant. As is the case for database recovery, the recovery time is proportional to the interval between file system checkpoints.

While standard LFS recovery quickly brings the file system to a physically consistent state, it does not provide the same guarantees made by *fsck*. When *fsck* completes, not only is the file system in a consistent state, but the directory structure has been verified as well. The five passes of *fsck* are summarized in Table 2. For LFS to provide the same level of robustness as FFS, LFS must make many of the same checks. While LFS has no bit maps to rebuild, the verification of block pointers and directory structure and contents is crucial for the system to recover from media failure. This recovery will be discussed in more detail in Section 3.4.

3. Engineering LFS

While the Sprite-LFS implementation was an excellent proof of concept, it had several deficiencies that made it unsuitable for a production environment. Our goal was to engineer a version of LFS that could be used as a replacement for FFS. Some of our concerns were as follows:

1. Sprite-LFS consumes excessive amounts of memory.
2. Write requests are successful even if there is insufficient disk space.
3. Recovery does nothing to verify the consistency of the file system directory structure.
4. Segment validation is hardware dependent.
5. All file systems use a single cleaner and a single cleaning policy.
6. There are no performance numbers that measure the cleaner overhead.

The earlier description of LFS focused on the overall strategy of log-structured file systems. The rest of Section 3 discusses how BSD-LFS addresses the first five problems listed above. Section 4 addresses the implementation issues specific to integration in a BSD framework, and Section 5 presents the performance analysis. In most ways, the logical framework of Sprite-LFS is unchanged. We have kept the segmented log structure and the major support structures associated with the log, namely the inode map, segment usage table, and cleaner. However, to address the problems described above and to integrate LFS into a BSD system, we have altered nearly all of the details of implementation, including a few fundamental design decisions. Most notably, we have moved the cleaner into user space, eliminated the directory operation log, and altered the segment layout on disk.

3.1. Memory Consumption

Sprite-LFS assumes that the system has a large physical memory and ties down substantial portions of it. The following storage is reserved:

Two 64K or 128K staging buffers
Since not all devices support scatter/gather I/O, data is written in buffers large enough to allow the maximum transfer size supported by the disk controller, typically 64K or 128K. These buffers are allocated per file system from kernel memory.

One cleaning segment
One segment's worth of buffer cache blocks per file system are reserved for cleaning.

Two read-only segments
Two segments' worth of buffer cache blocks per file system are marked read-only so that they may be reclaimed by Sprite-LFS without requiring an I/O.

Buffers reserved for the cleaner
Each file system also reserves some buffers for the cleaner. The number of buffers is specified in the superblock and is set during file system creation. It specifies the minimum number of clean buffers that must be present in the cache at any point in time. On the Sprite cluster, the amount of buffer space reserved for 10 commonly used file systems was 37 megabytes.

One segment
This segment (typically one-half megabyte) is allocated from kernel memory for use by the cleaner. Since this one segment is allocated per system, only one file system per system may be cleaned at a time.

The reserved memory described above makes Sprite-LFS a very "bad neighbor" as kernel subsystems compete for memory. While memory continues to become cheaper, a typical laptop system has only three to eight megabytes of memory, and might very reasonably expect to have three or more file systems.

BSD-LFS greatly reduces the memory consumption of LFS. First, BSD-LFS does not use separate buffers for writing large transfers to disk, instead it uses the regular buffer cache blocks. For disk controllers that do not coalesce contiguous reads, we use 64K staging buffers (briefly allocated from the regular kernel memory pool) to do transfers. The size of the staging buffer was set to the minimum of the maximum transfer sizes for currently supported disks. However, simulation results in [CAR92] show that for

current disks, the write size minimizing the read response time is typically about two tracks; two tracks is close to 64 kilobytes for the disks on our systems.

Secondly, rather than reserving read-only buffers, we initiate segment writes when the number of dirty buffers crosses a threshold. That threshold is currently measured in available buffer headers, not in physical memory, although systems with an integrated buffer cache and virtual memory will have simpler, more straight-forward mechanisms.

Finally, the cleaner is implemented as a user space process. This approach means that it requires no dedicated memory, competing for virtual memory space with the other processes.

3.2. Block Accounting

Sprite-LFS maintains a count of the number of disk blocks available for writing, i.e. the real number of disk blocks that do not contain useful data. This count is decremented when blocks are actually written to disk. This approach implies that blocks can be successfully written to the cache but fail to be written to disk if the disk becomes full before the blocks are actually written. Even if the disk is not full, all available blocks may reside in uncleaned segments and new data cannot be written. To prevent the system from deadlocking or losing data in these cases, BSD-LFS uses two forms of accounting.

The first form of block accounting is similar to that maintained by Sprite-LFS. BSD-LFS maintains a count of the number of disk blocks that do not contain useful data. It is decremented whenever a new block is created in the cache. Since many files die in the cache [BAKE91], this number is incremented whenever blocks are deleted, even if they were never written to disk.

The second form of accounting keeps track of how much space is currently available for writing. This space is allocated as soon as a dirty block enters the cache, but is not reclaimed until segments are cleaned. This count is used to initiate cleaning. If an application attempts to write data, but there is no space currently available for writing, the write will sleep until space is available. These two forms of accounting guarantee that if the operating system accepts a write request from the user, barring a crash, it will perform the write.

Accounting for the actual disk space required is difficult because inodes are not written into dirty buffers and segment summaries are not created until segments are written. Every time an inode is modified in the inode cache, a count of inodes to be written is incremented. When blocks are dirtied, the number of available disk blocks is decremented. To decide if there is enough disk space to allow another write into the cache, the number of segment summaries necessary to write what is in the cache is computed, added to the number of inode blocks necessary to write the dirty inodes and compared to the amount of space available on the disk. To create more available disk space, either the cleaner must run or dirty blocks in the cache must be deleted.

3.3. Segment Structure and Validation

Sprite-LFS places segment summary blocks at the end of segments trusting that if the write containing the segment summary is issued after all other writes in a partial segment, the presence of the segment summary validates the partial segment. This approach requires two assumptions: the disk controller will not reorder the write requests and the disk writes the contents of a buffer in the order presented. Since controllers often reorder writes and reduce rotational latency by beginning track writes anywhere on the track, we felt that BSD-LFS could not make these assumptions. We build segments from front to back, placing the segment summary at the beginning of each segment as shown in Figure 4. We compute a checksum across four bytes of each block in the partial segment, store it in the segment summary, and use this to verify that a partial segment is valid. This approach avoids write-ordering constraints and allows us to write multiple partial segments without an intervening seek or rotation. We do not yet have reason to believe that our checksum is insufficient, however, if it is, *patch tables* can be used to guarantee that any missing

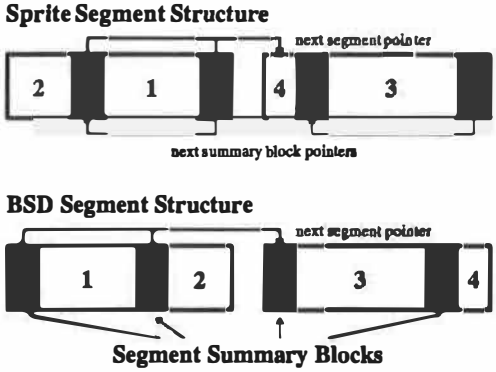


Figure 4: Partial Segment Structure Comparison Between Sprite-LFS and BSD-LFS. The numbers in each partial show the order in which the partial segments are created. Sprite-LFS builds segments back to front, chaining segment summaries. BSD-LFS builds segments front to back. After reading a segment summary block, the location of the next segment summary block can be easily computed.

sector can be detected during roll-forward, at the expense of a bit per disk sector stored in the segment usage table and segment summary blocks.

3.4. File System Verification

Fast recovery from system failure is desirable, but reliable recovery from media failure is necessary. Consequently, the BSD-LFS system provides two recovery strategies. The first quickly rolls forward from the last checkpoint, examining data written between the last checkpoint and the failure. The second does a complete consistency check of the file system to recover lost or corrupted data, due to the corruption of bits on the disk or errant software writing bad data to the disk. This check is similar to the functionality of *fsck*, the file system checker and recovery agent for FFS, and like *fsck*, it takes a long time to run.

As UNIX systems spend a large fraction of their time, while rebooting, in file system checks, the speed at which LFS is able to recover its file systems is considered one of its major advantages. However, FFS is an extremely robust file system. In the standard 4BSD implementation, it is possible to clear the root inode and recover the file system automatically with *fsck*(8). This level of robustness is necessary before users will accept LFS as a file system in traditional UNIX environments.

In terms of recovery, the advantage of LFS is that writes are localized, so the file system may be recovered to a physically consistent state very quickly. The BSD-LFS implementation permits LFS to recover quickly, and applications can start running as soon as the roll-forward has been completed, while basic sanity checking of the file system is done in the background. There is the obvious problem of what to do if the sanity check fails. It is expected that the file system will be forcibly made read-only, fixed, and then once again write enabled. These events should have a limited effect on users as it is unlikely to ever occur and is even more unlikely to discover an error in a file currently being written by a user, since the opening of the file would most likely have already caused a process or system failure. Of course, the root file system must always be completely checked after every reboot, in case a system failure corrupted it.

3.5. The Cleaner

In Sprite-LFS the cleaner is part of the kernel and implements a single cleaning policy. There are three problems with this, in addition to the memory issues discussed in Section 3.1. First, there is no reason to believe that a single cleaning algorithm will work well on all workloads. In fact, measurements in [SELT93b] show that coalescing randomly updated files would improve sequential read performance dramatically. Second, placing the cleaner in kernel-space makes it difficult to experiment with alternate cleaning policies. Third, implementing the cleaner in the kernel forces the kernel to make policy decisions (the cleaning algorithm) rather than simply providing a mechanism. To handle these problems, the BSD-LFS cleaner is implemented as a user process.

The BSD-LFS cleaner communicates with the kernel via system calls and the read-only *ifile*. Those functions that are already handled in the kernel (e.g. translating logical block numbers to disk addresses via *bmap*) are made accessible to the cleaner via system calls. If necessary functionality did not already exist in the kernel (e.g. reading and parsing segment summary blocks), it was relegated to user space.

There may be multiple cleaners, each implementing a different cleaning policy, running in parallel on a single file system. Regardless of the particular policy, the basic cleaning algorithm works as follows:

1. Read one or more target segments.
2. Decide which blocks are still alive.
3. Write live blocks back to the file system.
4. Mark the segment clean.

The *ifile* and four new system calls, summarized in Table 3, provide the cleaner with enough information to implement this algorithm. The cleaner reads the *ifile* to find out the status of segments in the file system and selects segments to clean based on this information. Once a segment is selected, the cleaner reads the segment from the raw partition and uses the first segment summary to find out what blocks reside in that partial segment. It constructs an array of BLOCK_INFO structures (shown in Figure 5) and continues scanning partial segments, adding their blocks to the array. When the entire segment has been read, and all the BLOCK_INFOs constructed, the cleaner calls *lfs_bmapv* which returns the current physical disk address for each BLOCK_INFO. If the disk address is the same as the location of the block in the segment being examined by the cleaner, the block is "live". Live blocks must to be written back into the file system without changing their access or modify times, so the cleaner issues an *lfs_markv* call, which is a special write causing these blocks to be appended into the log without updating the inode times.

Before rewriting the blocks, the kernel verifies that none of the blocks have "died" since the cleaner called *lfs_bmapv*. Once *lfs_markv* begins, only cleaned blocks are written into the log, until *lfs_markv* completes. Therefore, if cleaned blocks die after *lfs_markv* verifies that they are alive, partial segments written after the *lfs_markv* partial segments will reflect that the blocks have died. When *lfs_markv* returns, the cleaner calls *lfs_segclean* to mark the segment

lfs_bmapv	Take an array of inode number/logical block number pairs and return the disk address for each block. Used to determine if blocks in a segment are "live".
lfs_markv	Take an array of inode number/logical block number pairs and append them into the log. This operation is a special purpose write call that rewrites the blocks and inodes without updating the inode's access or modification times.
lfs_segwait	Causes the cleaner to sleep until a given timeout has elapsed or until another segment is written. This operation is used to let the cleaner pause until there may be more segments available for cleaning.
lfs_segclean	Mark a segment clean. After the cleaner has rewritten all the "live" blocks from a segment, the segment is marked clean for reuse.

Table 3: The System Call Interface for the Cleaner.

clean. Finally, when the cleaner has cleaned enough segments, it calls *lfs_segwait*, sleeping until the specified timeout elapses or a new segment is written into an LFS.

Since the cleaner is responsible for producing free space, the blocks it writes must get preference over other dirty blocks to be written to avoid running out of free space. There are degenerative cases where cleaning a segment can actually consume more space than it frees [SELT93a]. To ensure that the cleaner can always run and eventually generate more free space, normal writing is suspended when the number of clean segments drops to two.

The cleaning simulation results in [ROSE91] show that selection of segments to clean is an important design parameter in minimizing cleaning overhead, and that the cost-benefit policy defined there does extremely well for the simulated workloads. Briefly, each segment is assigned a cleaning *cost* and *benefit*. The *cost* to clean a segment is equal to:

$$1 + utilization$$

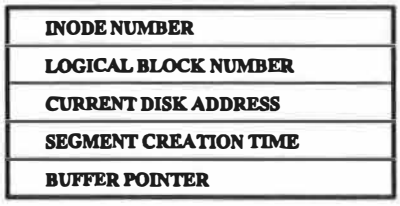


Figure 5: BLOCK_INFO Structure used by the Cleaner. The cleaner calculates the current disk address for each block from the disk address of the segment. The kernel specifies which have been superceded by more recent versions.

where utilization is the fraction of "live" data in the segment. The *benefit* of cleaning a segment is:

$$free\ bytes\ generated * age\ of\ segment$$

where *free bytes generated* is the fraction of "dead" blocks in the segment ($1 - utilization$) and *age of segment* is the time since the most recent modification to any block in that segment. When the file system needs to reclaim space, the cleaner selects the segment with the largest benefit to cost ratio. We retained this policy as the default cleaning algorithm.

Currently the cost-benefit cleaner is the only cleaner we have implemented, but two additional policies are under consideration. The first would run during idle periods and select segments to clean based on coalescing and clustering files. The second would flush blocks in the cache to disk during normal processing even if they were not dirty, if it would improve the locality for a given file. These policies will be analyzed in future work.

4. Implementing LFS in a BSD System

While the last section focused on those design issues that addressed problems in the design of Sprite-LFS, this section presents additional design issues either inherent to LFS or resulting from the integration of an LFS into 4BSD.

4.1. Integration with FFS

The on-disk data structures used by BSD-LFS are nearly identical to the ones used by FFS. This decision was made for two reasons. The first one was that many applications have been written over the years to interpret and analyze raw FFS structures. It is desirable that these tools could continue to function as before, with minor modifications to read the structures from a new location. The second and more important reason was that it was easy and increased the maintainability of the system. A basic LFS implementation,

without cleaner or reconstruction tools, but with *dumpfs*(1) and *newfs*(1) tools, was reading and writing from/to the buffer cache in under two weeks, and reading and writing from/to the disk in under a month. This implementation was done by copying the FFS source code and replacing about 40% of it with new code. The FFS and LFS implementations have since been merged to share common code.

In BSD and similar systems (e.g. SunOS, OSF/1), a file system is defined by two sets of interface functions, *vfs* operations and *vnode* operations [KLEI86]. *Vfs* operations affect entire file systems (e.g. mount, unmount, etc.) while *vnode* operations affect files (open, close, read, write, etc.).

File systems could share code at the level of a vfs or vnode subroutine call, but they could not share the UNIX naming while implementing their own disk storage algorithms. To allow sharing of the UNIX naming, the code common to both the FFS and BSD-LFS was extracted from the FFS code and put in a new, generic file system module (UFS). This code contains all the directory traversal operations, almost all vnode operations, the inode hash table manipulation, quotas, and locking. The common code is used not only by the FFS and BSD-LFS, but by the memory file system [MCKU90] as well. The FFS and BSD-LFS implementations remain responsible for disk allocation and actual I/O.

In moving code from the FFS implementation into the generic UFS area, it was necessary to add seven new *vnode* and *vfs* operations. Table 4 lists the operations that were added to facilitate this integration and explains why they are different for the two file systems.

4.1.1. Block Sizes

One FFS feature that is not implemented in BSD-LFS is fragments. The original reason FFS had fragments was that, given a large block size (necessary to obtain contiguous reads and writes and to lower the data to meta-data ratio), fragments were required to minimize internal fragmentation (allocated space that does not contain useful data). LFS does not require large blocks to obtain contiguous reads and writes as it sorts blocks in a file by logical block number, writing them sequentially. Still, large blocks are desirable to keep the meta-data to data ratio low. Unfortunately, large blocks can lead to wasted space if many small files are present. Since managing fragments complicates the file system, we decided to allocate progressively larger blocks instead of using a block/fragment combination. This improvement has not yet been implemented but is similar to the restricted buddy simulated in [SELT91].

Vnode Operations	
blkatoff	Read the block at the given offset, from a file. The two file systems calculate block sizes and block offsets differently, because BSD-LFS does not implement fragments.
valloc	Allocate a new inode. FFS must consult and update bit maps to allocate inodes while BSD-LFS removes the inode from the head of the free inode list in the *ifile*.
vfree	Free an inode. FFS must update bit maps while BSD-LFS inserts the inode onto a free list.
truncate	Truncate a file from the given offset. FFS marks bit maps to show that blocks are no longer in use, while BSD-LFS updates the segment usage table.
update	Update the inode for the given file. FFS pushes individual inodes synchronously, while BSD-LFS writes them in a partial segment.
bwrite	Write a block into the buffer cache. FFS does synchronous writes while BSD-LFS puts blocks on a queue for writing in the next segment.
Vfs Operations	
vget	Get a vnode. FFS computes the disk address of the inode while BSD-LFS looks it up in the *ifile*.

Table 4: New Vnode and Vfs Operations. These routines allowed us to share 60% of the original FFS code with BSD-LFS.

4.1.2. The Buffer Cache

Prior to the integration of BSD-LFS into 4BSD, the buffer cache had been considered file system independent code. However, the buffer cache contains assumptions about how and when blocks are written to disk. First, it assumes that a single block can be flushed to disk, at any time, to reclaim its memory. There are two problems with this: flushing blocks a single block at a time would destroy any possible performance advantage of LFS, and, because of the modified meta-data and partial segment summary blocks, LFS may require additional memory to write.

Therefore, BSD-LFS needs to guarantee that it can obtain any additional buffers it needs when it writes a segment. To prevent the buffer cache from trying to flush a single BSD-LFS page, BSD-LFS puts its dirty buffers on the kernel LOCKED queue, so that the buffer cache cannot reclaim them. The number of buffers on the locked queue is compared against two variables, the *start write threshold* and *stop access threshold*, to prevent BSD-LFS from using up all the available buffers. This problem can be much more reasonably handled by systems with better integration of the buffer cache and virtual memory.

Second, BSD maintains a logical block cache, hashed by vnode and logical block number. In FFS, since indirect blocks do not have logical block numbers, they are hashed by the vnode of the device (the file that represents the disk partition) and the disk block number. Since LFS does not assign disk addresses until blocks are written to disk, indirect blocks have no valid addresses on which to hash. To solve this problem, the block name space had to incorporate meta-data block numbering. This naming is done by making block addresses be signed integers with negative numbers referencing indirect blocks, while zero and positive numbers reference data blocks. Figure 6 shows how the blocks are numbered. Singly indirect blocks take on the negative of the first data block to which they point. Doubly and triply indirect blocks take the next lower negative number of the singly or doubly indirect block to which they point. This approach makes it simple to traverse the indirect block chains in either direction, facilitating reading a block or creating indirect blocks. Sprite-LFS partitions the "block name space" in a similar fashion. Although it is not possible for BSD-LFS to use FFS meta-data numbering, the reverse is not true. In 4.4BSD, FFS uses the BSD-LFS numbering and the *bmap* code has been moved into the UFS area.

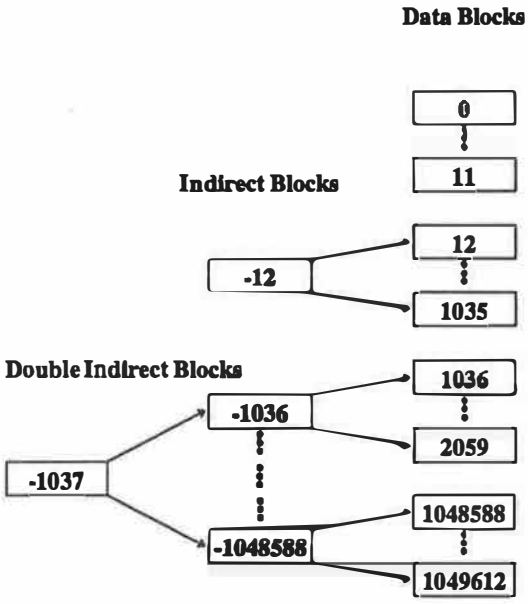


Figure 6: Block-numbering in BSD-LFS. In BSD-LFS, data blocks are assigned positive block numbers beginning with 0. Indirect blocks are numbered with the negative of the first data block that they address. Double and triple indirect blocks are numbered with one less than the first indirect or double indirect block that they address.

4.2. The IFILE

Sprite-LFS maintained the inode map and segment usage table as kernel data structures which are written to disk at file system checkpoints. BSD-LFS places both of these data structures in a read-only regular file, visible in the file system, called the *ifile*. There are three advantages to this approach. First, while Sprite-LFS and FFS limit the number of inodes in a file system, BSD-LFS has no such limitation, growing the *ifile* via the standard file mechanisms. Second, it can be treated identically to other files, in most cases, minimizing the special case code in the operating system. Finally, as is discussed in Section 3.5, we intended to move the cleaner into user space, and the *ifile* is a convenient mechanism for communication between the operating system and the cleaner. A detailed view of the *ifile* is shown in Figure 7.

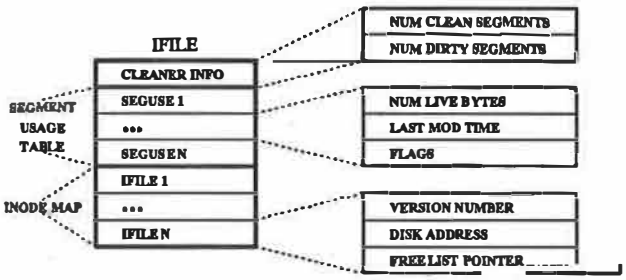


Figure 7: Detail Description of the IFILE. The *ifile* is maintained as a regular file with read-only permission. It facilitates communication between the file system and the cleaner.

Both Sprite-LFS and BSD-LFS maintain disk addresses and inode version numbers in the inode map. The version numbers allow the cleaner to easily identify groups of blocks belonging to files that have been truncated or deleted. Sprite-LFS also keeps the last access time in the inode map to minimize the number of blocks that need to be written when a file system is being used only for reading. Since the access time is eight bytes in 4.4BSD and maintaining

it in the inode map would cause the *ifile* to grow by 67%, BSD-LFS keeps the access time in the inode.

Sprite-LFS clusters inodes in the inode map, and allocates new inodes by picking a starting point and scanning forward sequentially until it finds a free inode. To create a new file, the inode map is searched from the inode entry of the containing directory. If a directory is being created, a random location is chosen. When a directory contains many files this scan is costly. On six Sprite file systems, the average number of entries searched per directory or file creation ranged from 26 to 192, with an average across all the file systems of 94 entries per allocation. BSD-LFS avoids this scan by maintaining a free list of inodes in the inode map.

The segment usage table contains the number of live bytes in and the last modified time of the segment, and is largely unchanged from Sprite-LFS. In order to support multiple and user mode cleaning processes, we have added a set of flags indicating whether the segment is clean, contains a superblock, is currently being written to, or is eligible for cleaning.

4.3. Directory Operations

Directory operations[1] pose a special problem for LFS. Since the basic premise of LFS is that operations can be postponed and coalesced to provide large I/Os, it is counterproductive to retain the synchronous behavior of directory operations. At the same time, if a file is created, filled with data and *fsync*ed, then both the file's data and the directory entry for the file must be on disk. Additionally, the UNIX semantics of directory operations are defined to preserve ordering (i.e. if the creation of file *a* precedes the creation of file *b*, then any post-recovery state of a file system that includes file *b* must include file *a*). We believe this semantic is used in UNIX systems to provide mutual exclusion and other locking protocols[2].

Sprite-LFS preserves the ordering of directory operations by maintaining a directory operation log inside the file system log. Before any directory updates are written to disk, a log entry that describes the directory operation is written. The log information always appears in an earlier segment, or the same segment, as the actual directory updates. At recovery time, this log is read and any directory operations that were not fully completed are rolled forward. Since this approach requires an additional, on-disk data structure, and since LFS is itself a log, we chose a different solution, namely *segment batching*.

Since directory operations affect multiple inodes, we need to guarantee that either both of the inodes and associated changes get written to disk or neither does. BSD-LFS has a unit of atomicity, the partial segment, but it does not have a mechanism that guarantees that all inodes involved in the same directory operation will fit into a single partial segment. Therefore, we introduced a mechanism that allows operations to span partial segments. At recovery, we never roll forward a partial segment if it has an unfinished directory operation and the partial segment that completes the directory operation did not make it to disk.

The requirements for segment batching are defined as follows:

1. If any directory operation has occurred since the last segment was written, the next segment write will append all dirty blocks from the *ifile* (that is, it will be a checkpoint, except that the superblock need not be updated).
2. During recovery, any writes that were part of a directory operation write will be ignored unless the entire write completed. A completed write can be identified if all dirty blocks of the *ifile* and its inode were successfully written to disk.

This definition is essentially a transaction where the writing of the *ifile* inode to disk is the commit operation. In this way, there is a coherent snapshot of the file system at some point after each directory operation. The penalty is that checkpoints are written more frequently in contrast to Sprite-LFS's approach that wrote additional logging information to disk.

The BSD-LFS implementation requires synchronizing directory operations and segment writing. Each time a directory operation is performed, the affected vnodes are marked. When the segment writer builds a segment, it collects vnodes in two passes. In the first pass, all unmarked vnodes (those not participating in directory operations) are collected, and during the second pass those vnodes that are marked are collected. If any vnodes are found during the second pass, this means that there are directory operations present in the current segment, and the segment is marked, identifying it as containing a directory operation. To prevent directory operations from being partially reflected in a segment, no new directory operations are begun while the segment writer is in pass two, and the segment writer cannot begin pass two while any directory operation is in progress.

[1] Directory operations include those system calls that affect more than one inode (typically a directory and a file) and include: create, link, mkdir, mknod, remove, rename, rmdir, and symlink.

[2] We have been unable to find a real example of the ordering of directory operations being used for this purpose and are considering removing it as unnecessary complexity. If you have an example where ordering must be preserved across system failure, please send us email at margo@das.harvard.edu!

When recovery is run, the file system can be in one of three possible states with regard to directory operations:

1. The system shut down cleanly so that the file system may be mounted as is.
2. There are valid segments following the last checkpoint and the last one was a completed directory-operation write. Therefore, all that is required before mounting is to rewrite the superblock to reflect the address of the *ifile* inode and the current end of the log.
3. There are valid segments following the last checkpoint or directory operation write. As in the previous case, the system recovers to the last completed directory operation write and then rolls forward the segments from there to either the end of the log or the first segment beginning a directory operation that is never finished. Then the recovery process writes a checkpoint and updates the superblock.

While rolling forward, two flags are used in the segment summaries: SS_DIROP and SS_CONT. SS_DIROP specifies that a directory operation appears in the partial segment. SS_CONT specifies that the directory operation spans multiple partial segments. If the recovery agent finds a segment with both SS_DIROP and SS_CONT set, it ignores all such partial segments until it finds a later partial segment with SS_DIROP set and SS_CONT unset (i.e. the end of the directory operation write). If no such partial segment is ever found, then all the segments from the initial directory operation on are discarded. Since partial segments are small [BAKE92] this should rarely, if ever, happen.

4.4. Synchronization

To maintain the delicate balance between buffer management, free space accounting and the cleaner, synchronization between the components of the system must be carefully managed. Figure 8 shows each of the synchronization relationships. The cleaner is given precedence over all other processing in the system to guarantee that clean segments are available if the file system has space. It has its own event variable on which it waits for new work (*lfs_allclean_wakeup*). The segment writer and user processes will defer to the cleaner if the disk system does not have enough clean space. A user process detects this condition when it attempts to write a block but the block accounting indicates that there is no space available. The segment writer detects this condition when it attempts to begin writing to a new segment and the number of clean segments has reached two.

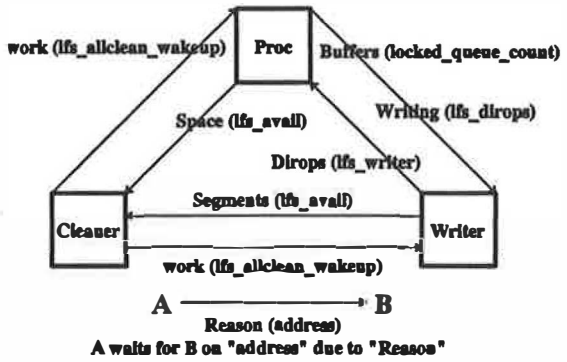


Figure 8: Synchronization Relationships in BSD-LFS. The cleaner has precedence over all components in the system. It waits on the *lfs_allclean_wakeup* condition and wakes the segment writer or user processes using the *lfs_avail* condition. The segment writer and user processes maintain directory operation synchronization through the *lfs_dirop* and *lfs_writer* conditions. User processes doing writes wait on the *locked_queue_count* when the number of dirty buffers held by BSD-LFS exceeds a system limit.

In addition to cleaner synchronization, the segment writer and user processes synchronize on the the availability of buffer headers. When the number of buffer headers drops below the *start write threshold* a segment write is initiated. If a write request would push the number of available buffer headers below the *stop access threshold*, the writing process waits until a segment write completes, making more buffer headers available. Finally, there is the directory operation synchronization. User processes wait on the *lfs_dirop* condition and the segment writer waits on *lfs_writer* condition.

4.5. Minor Modifications

There are a few additional changes to Sprite-LFS. To provide more robust recovery we replicate the superblock throughout the file system, as in FFS. Since the file system meta-data is stored in the *ifile*, we have no need for separate checkpoint regions, and simply store the disk address of the *ifile* inode in the superblock. Note that it is not necessary to keep a duplicate *ifile* since it can be reconstructed from segment summary information, if necessary.

5. Performance Measurements

This chapter compares the performance of the redesigned log-structured file system to more traditional file systems on a variety of benchmarks based on real workloads. The new log-structured file system was written in November of 1991 and was left largely untouched until late spring 1992 and is a completely

untuned implementation. While design decisions took into account the expected performance impact, at this point there is little empirical evidence to support those decisions.

The file systems against which LFS is compared are the regular fast file system (FFS), and an enhanced version of FFS similar to that described in [MCVO91], referred to as EFS for the rest of this paper.

5.1. Extent-like Performance from FFS

EFS provides extent-based file system behavior without changing the underlying structures of FFS, by allocating blocks sequentially on disk and clustering multiple block requests. FFS is parameterized by a variable called *maxcontig* that specifies how many logically sequential disk blocks should be allocated contiguously. When maxcontig is large (equal to a track), FFS does what is essentially track allocation. In EFS, sequential dirty buffers are accumulated in the cache, and when an extent's worth (i.e. *maxcontig* blocks) have been collected, they are bundled together into a cluster, providing extent-based writing.

To provide extent-based reading, the interaction between the buffer cache and the disk was modified. Typically, before a block is read from disk, the *bmap* routine is called to translate logical block addresses to physical disk block addresses. The block is then read from disk and the next block is requested. Since I/O interrupts are not handled instantaneously, the disk is usually unable to respond to two contiguous requests on the same rotation, so sequentially allocated blocks incur the cost of an entire rotation. For both EFS and BSD-LFS, *bmap* was extended to return, not only the physical disk address, but the number of contiguous blocks that follow the requested block. Then, rather than reading one block at a time and requesting the next block asynchronously, the file system reads many contiguous blocks in a single request, providing extent-based reading. Because BSD-LFS potentially allocates many blocks contiguously, it may miss rotations between reading collections of blocks. Since EFS uses the FFS allocator, it leaves a rotational delay between clusters of blocks and does not pay this penalty.

5.2. The Evaluation Platform

Our benchmarking configuration consisted of a Hewlett-Packard series 9000/380 computer with a 25 Mhz MC68040 processor. It had 16 megabytes of main memory, and an HP 97560 SCSI disk. The hardware configuration is summarized in Table 5. The system was running the 4.4BSD-Alpha operating system and all measurements were taken with the system running single-user, unattached to any network. Each of the file systems used a 4-kilobyte block size with FFS and EFS using 1-kilobyte fragments.

The three file systems being evaluated run in the same operating system kernel and share most of their source code. There are approximately 6000 lines of shared C code, 4000 lines of LFS-specific code, and 3500 lines of FFS-specific code. EFS uses the same source code as FFS plus an additional 500 lines of clustering code, of which 300 are also used by BSD-LFS (for reads).

Each of the next sections describes a benchmark and presents performance analysis for each file system. The first benchmark analyzes raw file system performance. The next two benchmarks emulate specific workloads. A time-sharing environment is represented by a software development benchmark, and a database environment is represented by a modified version of the industry-standard TPC-B benchmark [TPCB90]. For a more thorough analysis of LFS and a wider range of benchmarks, see [SELT93a].

5.2.1. Raw File System Performance

The goal of this test is to measure the maximum throughput that can be expected from the given disk and system configuration for each of the file systems. For this test, the three file systems are compared

Disk (SCSI HP 97560)	
Average seek	13.0 ms
Single rotation	15.0 ms
Track size	36 KB
Track buffer	128 KB
Disk bandwidth	2.2 MB/sec
Bus bandwidth	1.6 MB/sec
Controller overhead	1.0 ms
Track skew	8 sectors
Cylinder skew	10 sectors
Cylinder size	19 tracks
Disk size	1962 cylinders
CPU (Motorola 68040)	
Memory Bandwidth	12.0 MB/sec
CPU	25 Mhz
MIPS	10-12

Table 5: Hardware Specifications. Although the disk can transfer at 2.2 megabytes per second, the bus bandwidth is limited to 1.6 megabytes per second. SCSI supports two transfer modes, synchronous and asynchronous [ADAP85]. Synchronous mode is optional under SCSI-I and is not supported by the disk driver. Therefore, all transfers are performed using asynchronous mode and are limited to 1.6 MB/sec.

against the maximum speed at which the operating system can write directly to the disk. The benchmark consists of creating a file of size *S* and then either reading or writing the entire file 50 times. The measurements recorded are averages across the 50 runs. For the read tests, the cache is flushed before each test by unmounting and remounting the file system.

Raw Write Performance

The graph in Figure 9 shows the bandwidth attained for writing, as a function of *S*, the size of the I/O. Given the sequential layout of both LFS and EFS, the expectation is that both should perform comparably to the speed of the raw disk and that FFS, with its rotational positioning, should achieve approximately 50% of the disk bandwidth. However, there are several anomalies.

First, as the I/O size increases, EFS actually provides more bandwidth than the raw disk partition. The explanation for this can be found by looking at the number of synchronous I/O's and the begin time for each operation. When accessing the raw partition, all I/Os are synchronous. Therefore, there is no overlap between the time required to copy the data from user-space into the kernel and the time required to perform the I/Os. As a result, there is a gap of approximately five milliseconds between the completion of each I/O and the initiation of the next I/O. In contrast, EFS has an aggressive buffering policy, allowing it to perform asynchronous writes in units of 64 kilobytes. Therefore, the I/Os are queued, and successive I/Os are begun almost immediately.

The next anomaly lies in the fact that LFS performs noticeably worse than either EFS or the RAW partition. This is an artifact of this benchmark as opposed to a fundamental difference in the attainable write bandwidth of the two file systems. The problem is that the benchmark performs the write and then calls *fsync* to ensure that the blocks have been written to disk.

LFS achieves its write performance by buffering a large number of dirty buffers before initiating I/O. As a result, LFS does not begin writing any data to disk until requested to do so by the application *fsync* or until the *start write threshold* has been reached. On this system, the *start write threshold* results in approximately 800 kilobytes of data being buffered before a write is initiated. As a result, for all the tests where the transfer size was smaller than one megabyte, the benchmark had two phases, the first in which data was written into the cache, and the second during which time the data was being written to disk.

To verify this, timings were taken after all the writes had been issued, but before the call to *fsync* and then again after the call to *fsync*. In the tests where the total transfer size was less than 800K, LFS' elapsed time for the *fsync* was nearly identical to the time required for EFS to write all its buffers. Figure 10 depicts this behavior. In the tests where the transfer size was greater than 800K, LFS' elapsed time for the *fsync* was the time reported for the synchronous LFS write that flushed the data remaining in the cache at the time of the *fsync*.

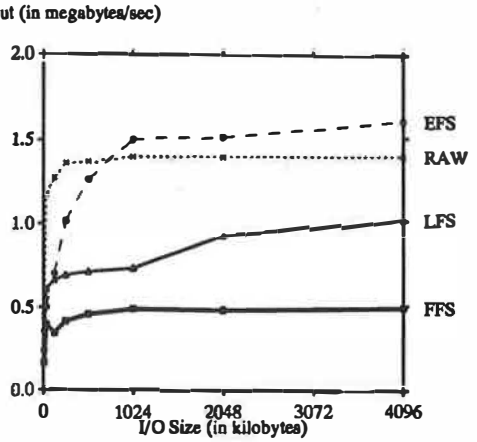


Figure 9: Maximum File System Write Bandwidth. This graph shows the write bandwidth of each file system as a function of the transfer size. EFS attains the best performance, as it performs nearly all its writes asynchronously in maximal-sized buffers. Writes to the RAW partition also occur in maximal-sized units, but are performed synchronously. In LFS, since a large amount of data is gathered in the cache before being written to the disk, there is less overlap between CPU processing and disk activity, leading to the gap shown above. The rotational delay of FFS prohibits it from achieving more than 25% of the available disk bandwidth.

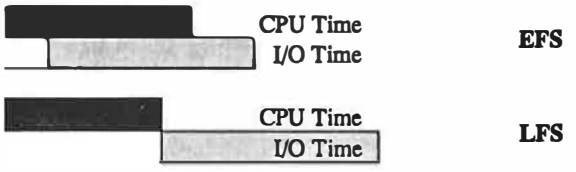


Figure 10: Effects of LFS Write Accumulation. The bars represent elapsed time for each phase of the benchmark on a one-half megabyte write. EFS effectively overlaps I/O and CPU processing while LFS waits until all the data is accumulated before initiating the write. As a result, the bandwidth measured by this test appears much lower for LFS.

These write tests were repeated for LFS with the cleaner running, but the results were indistinguishable from the results without the cleaner. Since the same data is overwritten for each iteration of the test, there are always empty segments available for reclamation by the cleaner. As a result, the cleaner reported that it always cleaned empty segments, and the overhead was unmeasurable.

The last anomaly is that FFS did not achieve the 50% bandwidth expected, but achieved closer to 31% of the transfer bandwidth (0.5 megabytes per second of the possible 1.6 megabytes per second). The explanation of this is in the FFS *rot_delay* parameter. This parameter is used by FFS to express the length of time, from the disk's perspective, that it takes the CPU to acknowledge the completion of an I/O and to issue another one.[3]

For the system under test, the *rot_delay* that provided the best performance was experimentally determined to be 4 milliseconds. This value was determined by building file systems with successively larger *rot_delays* and selecting the value that led to the best performance. However, with a rotational latency of 15 milliseconds, 36 kilobyte tracks, 4-kilobyte blocks, and a 4 millisecond *rot_delay*, only one in four blocks is allocated to the same file, as shown in Figure 11. The maximum transfer bandwidth of the disk is is 2.2 megabytes per second and one-quarter of this is 0.55 megabytes per second, which is close to the observed performance of FFS.

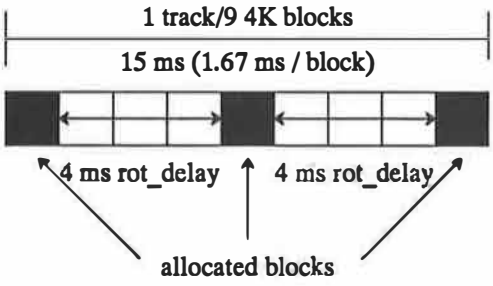


Figure 11: Impact of Rotational Delay on FFS Performance. Since *rot_delay* for this disk is 4 milliseconds, FFS will allocate only one in every four blocks. Therefore, at most 3 blocks (2.25 on average) can be accessed on each disk rotation. Therefore, FFS will attain at most one-quarter of the maximum bandwidth of the disk.

[3] This is based on the assumption that queueing is performed by the host and not the disk.

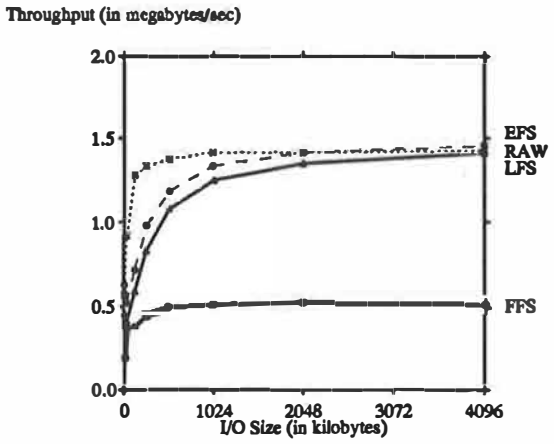


Figure 12: Maximum File System Read Bandwidth. The graph shows the maximum read throughput attained by each file system as a function of the transfer size. As EFS and LFS allocate blocks contiguously and use the exact same read-ahead algorithm, the expectation is that both will perform comparably to the raw partition. Once again, FFS is limited to approximately 25% of the total disk bandwidth due to rotational delays between allocated blocks.

Raw Read Performance

The results of the raw read tests, shown in Figure 12, are much closer to what is expected. FFS demonstrates read performance nearly identical to its write performance since it is limited by the number of blocks transferred during a single rotation. Both LFS and EFS perform comparably to the raw disk with very small (3%) differences in performance.

This benchmark demonstrates that both EFS and LFS can utilize close to 100% of the available I/O bandwidth on large I/Os. When individual write response time is an issue, LFS incurs a performance penalty due to its delayed write policy.

The remaining tests are all designed to stress the file systems. For BSD-LFS, that means the cleaner is running and the disk partitions are 80% utilized, so that the cleaner is forced to reclaim space. For EFS and FFS, as the disk partition fills, it becomes more difficult for them to allocate blocks optimally.

5.2.2. Software Development Workload

The next tests evaluate BSD-LFS in a typical software development environment. The Andrew benchmark [HOWA88] is often used for this type of measurement. It contains five phases:

1. Create a directory hierarchy.

2. Make a copy of the data.
3. Recursively examine the status of every file.
4. Examine every byte of every file.
5. Compile several of the files.

Unfortunately, the test set for the Andrew benchmark is small, and main-memory file caching can make the results uninteresting. In order to exercise the file systems, this benchmark is run both single-user and multi-user (where several invocations of the benchmark are run concurrently).

Single-User Andrew Performance

Table 6 shows the performance of the original Andrew benchmark. The entire five-phase test was run ten times for each of FFS, EFS, and LFS, with the directory hierarchy deleted after each pass. For the LFS test with the cleaner running (LFSC), the test was repeated 100 times to ensure that the file system was completely overwritten at least twice. In order to understand the differences in performance, kernel counters that record disk and LFS statistics were initialized before, and sampled after, each phase.

Overall, LFS demonstrates a 9% improvement over EFS and FFS, which perform comparably. The difference is isolated to phases one, two, and five. It is not surprising that LFS would outperform the other systems in phase one, the create phase, as LFS performs all its directory creations asynchronously, performing no writes, while EFS and FFS issue 100 synchronous writes each. As phase two is the write-intensive phase, it is also expected that LFS will perform better, and it does so, demonstrating 37% better performance than the other two systems. Again, EFS and FFS are performing a great deal of I/O (263 requests for about 750 kilobytes), over half of which are synchronous (as a result of closing files). LFS performs no writes during this phase as all the data is written to the cache.

Phase 5, which is moderately CPU-intensive (59% CPU utilization for LFS and 49% for EFS), surprisingly demonstrates a small (3-5%) advantage for LFS. Once again, kernel disk counters reveals that EFS and FFS are synchronously writing the output object files to disk (45 of 48 writes) while LFS is buffering the data and performing nearly one-third the number of writes.

The more striking difference is in the number of reads issued by the two file systems in phase five. LFS issues only a single read while FFS issues 46 of them. The explanation for this also lies in file allocation. When FFS creates a file, it allocates an inode from the appropriate cylinder group and then reads the contents of the inode from disk. (This is an artifact of the file system architecture and could be avoided by modifying the interface to the *vfs* routine *vfs_vget*.) In LFS, new inodes are created in memory, not read from the disk.

These single-user results differ slightly from those presented in [ROSE92]. First, the compilation phase in [ROSE92] is much longer than in this test because different compilers were used. Secondly, the results in [ROSE92] show LFS providing a 40% performance improvement on phase 3 (the phase that examines every inode) and a 29% performance improvement on phase 4 (the phase that examines every byte), while the results here show virtually no difference. Phases 3 and 4 perform no I/O on any of the file systems, so performance is limited strictly by the file system code that reads data from the cache, traverses directories, and reads inodes from the in-memory inode cache. Since the three file systems share the same code for performing these functions, the expectation is that the systems should behave identically. Since the system measured in [ROSE92] is unavailable for instrumentation, it is unclear why results on phases 3 and 4 differ.

Multi-User Andrew Performance

The multi-user version of Andrew shows the file system performance as a function of the degree of

	Phase 1 Create Directories	Phase 2 Copy Files	Phase 3 Stat Touch Inodes	Phase 4 Grep Touch Bytes	Phase 5 Compile	Total
FFS	2.10 (0.30)	7.90 (0.30)	6.30 (0.46)	9.00 (0.00)	44.80 (0.40)	70.1 (0.70)
EFS	2.10 (0.30)	7.90 (0.30)	6.70 (1.19)	9.10 (0.30)	44.40 (0.49)	70.2 (1.60)
LFS	0.33 (0.47)	5.00 (0.00)	6.50 (0.81)	9.07 (0.25)	42.90 (1.40)	63.8 (2.34)
LFSC	0.43 (0.49)	5.09 (0.28)	6.37 (0.48)	9.07 (0.26)	42.61 (0.49)	63.6 (0.62)

Table 6: Single-User Andrew Benchmark Results. This table shows the elapsed time for each phase of the benchmark on each file system. Reported times are the average across ten iterations with the standard deviation in parentheses. LFSC indicates that the benchmark was run on the log-structured file system with the cleaner running, but the similarity in results for most phases indicates that the cleaner had virtually no impact on performance. Overall, LFS demonstrates approximately a 9% difference in performance which can be attributed to asynchronous file creation and write-clustering.

multiprogramming. The test is performed by running N concurrent invocations of the benchmark, with each invocation creating, traversing, and removing its own directory hierarchy. The reported results are the averages of the results of each of ten runs for each invocation. The resulting averages are divided by the multiprogramming level to produce the metric "elapsed seconds per invocation."

The goal of the multi-user test is to examine two aspects of the file systems' behavior under the software development workload. First, as the multiprogramming level increases, the entire data set no longer fits in the cache, so more I/O is performed. Secondly, with separate directory hierarchies, the different forms of locality used by LFS (temporal locality -- files created at about the same time reside close together) and FFS (logical locality -- files in the same directory are placed close together) can be compared.

The multi-user performance is the result of two competing factors. As concurrent invocations of the benchmark compete for resources, the utilization of both the CPU and the disk increases, as does performance. However, after the multiprogramming level exceeds two, the total working set becomes too large to fit in the cache and the total I/O time increases. Towards the left-hand side of the graph, the predominant factor is the overlap between the CPU and the disk. As LFS is already performing most of its I/O asynchronously, it has less room for improvement than EFS and FFS. So, the CPU utilization for LFS increases from 60% to 80% while the CPU utilization for EFS goes from 50% to 89%, explaining the steeper decline in elapsed time for EFS than for LFS.

As the multiprogramming level exceeds four, the data sets no longer fit in the cache and the read performance becomes the dominant factor for all the file systems. Kernel I/O statistics reveal that, on average, LFS is performing more seeks than EFS, explaining the small difference in performance observed as the multiprogramming level increases.

This benchmark indicates that LFS and EFS perform comparably on this particular software development workload. To generalize, LFS demonstrates superior file creation performance, but logical locality appears better than temporal locality when the working set is too large to fit in the cache. The next benchmark demonstrates this even more dramatically.

5.2.3. Transaction Processing Performance

A modified version of the industry-standard TPC-B is used as the database-oriented benchmark.

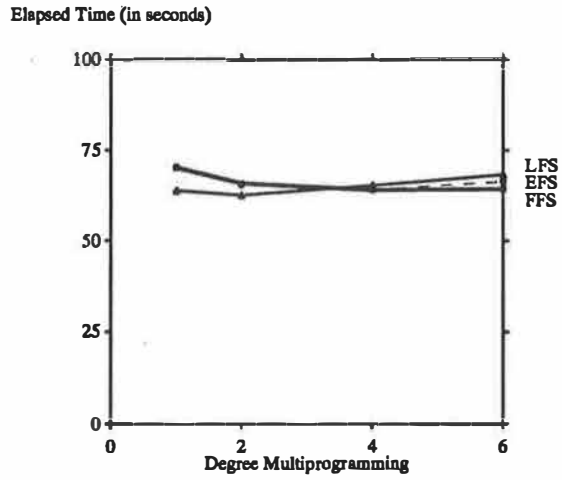


Figure 13: Multi-User Andrew Performance. This graph shows the elapsed time for all five phases of the Andrew benchmark under increasing multiprogramming. Overall, the impact of multiprogramming is less significant than might have been expected, yielding at most a 9% performance improvement.

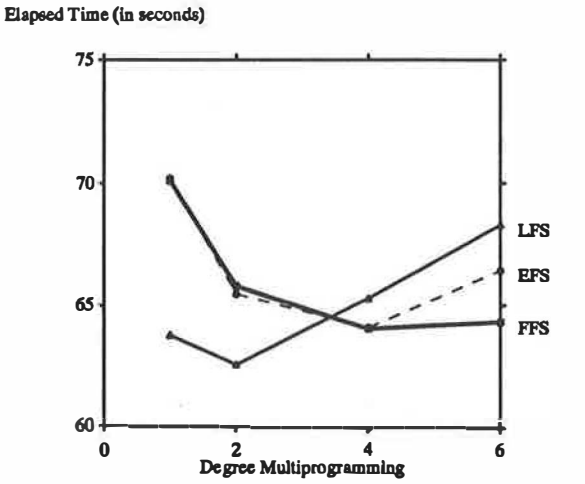


Figure 14: Multi-User Andrew Performance (Blow-Up). This graph emphasizes the small performance differences in the multi-user Andrew benchmark. For EFS and FFS which perform many synchronous operations, multi-programming allows the overlapping of CPU and disk and reduces per-invocation time. LFS also benefits from this overlap, but not as significantly as the other systems. The second effect is that the total data set size begins to exceed the cache capacity and read performance becomes the dominant factor.

The system is configured for a ten transaction-per-second system, but runs single-user without a redundant log and does not model think time. Each measurement in Table 7 represents ten runs of 1000 transactions. The counting of transactions is not begun until the buffer pool has filled, so the measurements do not exhibit any artifacts of an empty cache. Transaction run lengths of greater than 1000 were measured, but there was no noticeable change in performance after the first 1000 transactions.

When the cleaner is not running, LFS provides a 15% performance improvement over EFS. However, the impact of the cleaner is far worse than was anticipated. The benchmark randomly updates blocks in the 237 megabyte account relation, leaving most segments fairly full. During the course of the benchmark, the cleaner cleaned approximately one segment for every 50 transactions executed. On average, the cleaned segments were 71% utilized and cleaner writes accounted for between 60% and 80% of the total blocks written and 31% of all blocks transferred.

In an attempt to reduce cleaner overhead, a second set of tests were run with a smaller segment size (256 kilobytes). The performance before cleaning is the same as for the one megabyte case, but the after-cleaning performance is only slightly better (about 6%). As in the one megabyte case, the majority of the writes performed are on behalf of the cleaner (60-70%). While the smaller segment size reduces the variation in response time as evidenced through the smaller standard deviation, it does not significantly improve performance as most of the write activity is due to the cleaner. Although a user-level cleaner avoids synchronization costs between user processes and the kernel, it cannot avoid contention on the disk arm.

	Transactions per second	Elapsed Time 1000 transactions
FFS	14.2	70.23 (1.0%)
EFS	16.8	59.51 (2.1%)
LFS (no cleaner)	19.3	51.75 (0.6%)
LFS (cleaner, 1M)	11.6	85.86 (5.3%)
LFS (cleaner, 256 K)	12.4	80.72 (1.8%)

Table 7: Modified TPC-B Performance Results. The test database was scaled for a 10 transaction-per-second system (1,000,000 accounts, 100 tellers, and 10 branches). The elapsed time and standard deviation, as a percent of the elapsed time, is reported for runs of 1000 transactions. The LFS results show performance before the cleaner begins to run and after the cleaner begins to run. Since the cleaner decreased performance by 40%, a second test was run with 256 kilobyte segments. Even with the smaller segment size, the cleaner decreased performance by 35%.

6. Conclusions

The implementation of BSD-LFS highlighted some subtleties in the overall LFS strategy as well as some performance deficiencies. While LFS can utilize a large fraction of the disk bandwidth for writing, the cleaner has a severe impact in certain workloads, particularly transaction processing.

While allocation in BSD-LFS is simpler than in extent-based file systems or file systems like FFS, the management of memory is much more complicated. The Sprite-LFS implementation addressed this problem by reserving large amounts of memory. Since this is not feasible in most environments, a more complex mechanism to manage buffer and memory requirements is necessary. LFS operates best when it can write out many dirty buffers at once. However, holding dirty data in memory until much data has accumulated requires consuming more memory than might be desirable and may not be allowed (e.g. NFS semantics require synchronous writes). In addition, the act of writing a segment requires allocation of additional memory (for segment summaries and on-disk inodes), so segment writing needs to be initiated before memory becomes a critical resource to avoid memory thrashing or deadlock.

The delayed allocation of BSD-LFS makes accounting of available free space more complex than that in a pre-allocated system like FFS. In Sprite-LFS, the space available to a file system is the sum of the disk space and the buffer pool. As a result, data is written to the buffer pool for which there might not be free space available on disk. Since the applications that wrote the data may have exited before the data is written to disk, there is no way to report the "out of disk space" condition. This failure to report errors is unacceptable in a production environment. To avoid this phenomena, available space accounting must be done as dirty blocks enter the cache instead of when they are written from cache to disk.

7. Future Directions

The novel structures of BSD-LFS makes it an exciting vehicle for adding functionality to the file system. For example, there are two characteristics of BSD-LFS that make it desirable for transaction processing. First, the multiple, random writes of a single transaction get bundled and written at sequential speeds, so we expect to see a dramatic performance improvement in multi-user transaction applications, if sufficient disk space is available. Second, since data is never overwritten, before-images of updated pages

exist in the file system until they are reclaimed by the cleaner. An implementation that exploits these two characteristics is described and analyzed in [SELT93b] on Sprite-LFS, and we plan on doing a prototype implementation of transactions in BSD-LFS.

The "no-overwrite" characteristic of BSD-LFS makes it ideal for supporting *unrm* which would undo a file deletion. Saving a single copy of a file is no more difficult than changing the cleaner policy to not reclaim space from the last version of a file, and the only challenge is finding the old inode. More sophisticated versioning should be only marginally more complicated.

Also, the sequential nature of BSD-LFS write patterns makes it nearly ideal for tertiary storage devices [KOHL93]. LFS may be extended to include multiple devices in a single file system. If one or more of these devices is a robotic storage device, such as a tape stacker, then the file system may have tremendous storage capacity. Such a file system would be particularly suitable for on-line archival or backup storage.

An early version of the BSD-LFS implementation was shipped as part of the 4.4BSD-Alpha release. The current version described in this paper will be available as part of 4.4BSD. Additionally, the FFS shipped with 4.4BSD will contain the enhancements to provide clustered reading and writing.

8. Acknowledgements

Mendel Rosenblum and John Ousterhout are responsible for the original design and implementation of Sprite-LFS. Without their work, we never would have tried any of this. We also wish to express our gratitude to John Wilkes and Hewlett-Packard for their support.

9. References

[ADAP85] *SCSI User Guide*, Adaptive Data Systems Inc., Pomona, CA, 1985.

[BAKE91] Baker, M., Hartman., J., Kupfer., M., Shirriff, L., Ousterhout, J., "Measurements of a Distributed File System," *Proceedings of the 13th Symposium on Operating System Principles*, Monterey, CA, October 1991, 198-212. Published as *Operating Systems Review 25*, 5 (October 1991).

[BAKE92] Baker, M., Asami, S., Deprit, E., Ousterhout, S., Seltzer, M., "Non-Volatile Memory for Fast, Reliable File Systems," *Proceedings of the Fifth Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, October 1992.

[CAR92] Carson, S., Setia, S., "Optimal Write Batch Size in Log-structured File Systems," *Proceedings of 1992 Usenix Workshop on File Systems*, Ann Arbor, MI, May 21-22 1992, 79-91.

[KAZA90] Kazar, M., Leverett, B., Anderson, O., Vasilis, A., Bottos, B., Chutani, S., Everhart, C., Mason, A., Tu, S., Zayas, E., "DECorum File System Architectural Overview," *Proceedings of the 1990 Summer Usenix* Anaheim, CA, June 1990, 151-164.

[HAER83] Haerder, T. Reuter, A. "Principles of Transaction-Oriented Database Recovery," *Computing Surveys*, 15(4); 1983, 237-318.

[HOWA88] Howard, J., Kazar, Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, N., West, M., "Scale and Performance in a Distributed File System," *ACM Transaction on Computer Systems 6*, 1 (February 1988), 51-81.

[KLEI86] S. R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," *Usenix Conference Proceedings*, June 1986, 238-247.

[KOHL93] Kohl, J., Staelin, C., Stonebraker, M., "Highlight: Using a Log-structured File System for Tertiary Storage Management," *Proceedings 1993 Winter Usenix*, San Diego, CA, January 1993.

[MCKU84] Marshall Kirk McKusick, William Joy, Sam Leffler, and R. S. Fabry, "A Fast File System for UNIX", *ACM Transactions on Computer Systems*, 2(3), August 1984, 181-197.

[MCKU90] Marshall Kirk McKusick, Michael J. Karels, Keith Bostic, "A Pageable Memory Based Filesystem," *Proceedings of the 1990 Summer Usenix Technical Conference*, Anaheim, CA, June 1990, 137-144.

[MORA90] Moran, J., Sandberg, R., Coleman, D., Kepecs, J., Lyon, B., Breaking Through the NFS Performance Barrier," *Proceedings of the 1990 Spring European Unix Users Group*, Munich, Germany, 199-206, April 1990.

[MCVO91] McVoy, L., Kleiman, S., "Extent-like Performance from a Unix File System," *Proceedings Winter Usenix 1991*, Dallas, TX, January 1991, 33-44.

[OUST85] Ousterhout, J., Costa, H., Harrison, D., Kunze, J., Kupfer, M., Thompson, J., "A Trace-Driven Analysis of the UNIX 4.2BSD File System," *Proceedings of the Tenth Symposium on Operating*

System Principles, December 1985, 15-24. Published as *Operating Systems Review 19*, 5 (December 1985).

[OUST89] Ousterhout, J., Douglis, F., "Beating the I/O Bottleneck: A Case for Log-structured File Systems," *Operating Systems Review 23*, 1, January 1989, 11-27. Also available as Technical Report UCB/CSD 88/467.

[ROSE90] Rosenblum, M., Ousterhout, J. K., "The LFS Storage Manager," *Proceedings of the 1990 Summer Usenix*, Anaheim, CA, June 1990, 315-324.

[ROSE91] Rosenblum, M., Ousterhout, J. K., "The Design and Implementation of a Log-Structured File System," *Proceedings of the Symposium on Operating System Principles*, Monterey, CA, October 1991, 1-15. Published as *Operating Systems Review 25*, 5 (October 1991). Also available as *Transactions on Computer Systems 10*, 1 (February 1992), 26-52.

[ROSE92] Rosenblum, M., "The Design and Implementation of a Log-structured File System," PhD Thesis, University of California, Berkeley, June 1992. Also available as Technical Report UCB/CSD 92/696.

[SELT90] Seltzer, M., Chen, P., Ousterhout, J., "Disk Scheduling Revisited," *Proceedings of the 1990 Winter Usenix*, Washington, D.C., January 1990, 313-324.

[SELT91] Seltzer, M., Stonebraker, M., "Read Optimized File Systems: A Performance Evaluation," *Proceedings 7th Annual International Conference on Data Engineering*, Kobe, Japan, April 1991, 602-611.

[SELT93a] Seltzer, M., "File System Performance and Transaction Support," PhD Thesis, University of California, Berkeley, May 1993.

[SELT93b] Seltzer, M., "Transaction Support in a Log-Structured File System," To appear in the *Proceedings of the 1993 International Conference on Data Engineering*, Vienna, Austria, April 1993.

[THOM78] Thompson, K., "Unix Implementation," *Bell Systems Technical Journal*, 57(6), part 2, July-August 1978, 1931-1946.

[TPCB90] Transaction Processing Performance Council, "TPC Benchmark B," Standard Specification, Waterside Associates, Fremont, CA., 1990.

Margo I. Seltzer is an Assistant Professor at Harvard University. Her research interests include file systems, databases, and transaction processing systems. She spent several years working at startup companies designing and implementing file systems and transaction processing software and designing microprocessors. Ms. Seltzer completed her Ph.D. in Computer Science at the University of California, Berkeley in December of 1992 and received her AB in Applied Mathematics from Harvard/Radcliffe College in 1983.

Keith Bostic has been a member of the Berkeley Computer Systems Research Group (CSRG) since 1986. In this capacity, he was the principle architect of the 2.10BSD release of the Berkeley Software Distribution for PDP-11's. He is currently one of the two principal developers in the CSRG, continuing the development of future versions of Berkeley UNIX. He received his undergraduate degree in Statistics and his Masters degree in Electrical Engineering from George Washington University. He is a member of the ACM, the IEEE and several POSIX working groups.

Dr. Marshall Kirk McKusick got his undergraduate degree in Electrical Engineering from Cornell University. His graduate work was done at the University of California, where he received Masters degrees in Computer Science and Business Administration, and a Ph.D. in the area of programming languages. While at Berkeley he implemented the 4.2BSD fast file system and was involved in implementing the Berkeley Pascal system. He currently is the Research Computer Scientist at the Berkeley Computer Systems Research Group, continuing the development of future versions of Berkeley UNIX. He is past-president of the Usenix Association, and a member of ACM and IEEE.

Carl Staelin works for Hewlett-Packard Laboratories in the Berkeley Science Center and the Concurrent Systems Project. His research interests include high performance file system design, and tertiary storage file systems. As part of the Science Center he is currently working with Project Sequoia at the University of California at Berkeley. He received his PhD in Computer Science from Princeton University in 1992 in high performance file system design.

Exploiting In-Kernel Data Paths to Improve I/O Throughput and CPU Availability

Kevin Fall & Joseph Pasquale[1] – University of California, San Diego

ABSTRACT

We present the motivation, design, implementation, and performance evaluation of a UNIX kernel mechanism capable of establishing fast in-kernel data pathways between I/O objects. A new system call, *splice()* moves data asynchronously and without user-process intervention to and from I/O objects specified by file descriptors. Performance measurements indicate improved I/O throughput and increased CPU availability attributable to data copying and context switch overhead.

Introduction

Improved computer hardware has enabled the development of complex applications with enormous I/O demands. Providing adequate performance for such applications poses a significant challenge to the operating systems community, especially with the growing popularity of multimedia applications and systems. Although both application demands and hardware performance have witnessed great gains in recent years, I/O system software performance has not received commensurate attention. Furthermore, fundamental assumptions in the I/O system structure may limit achievable performance by introducing unnecessary overheads.

I/O Intensive applications are those applications moving a large amount of data (on the order of hundreds or thousands of megabytes). Many applications, especially multimedia applications, require the movement of large volumes of data between devices or files in a timely fashion with minimal intermediate manipulation or processing. Concepts useful for improving I/O system performance for these applications include minimization of data movement within memory, and separating I/O control from I/O data transfer [Pas92].

With UNIX being the primary operating system available for most scientific and high performance computing platforms today, evaluation and improvement of UNIX system performance when exposed to I/O intensive workloads is important to ensure a viable future execution environment. This study outlines UNIX I/O system modifications aimed at improving the performance of such applications in current systems. We believe the mechanisms presented here to be generally applicable to other systems supporting a memory-based I/O buffer interface to user processes.

[1]This research was supported in part by grants from DEC, IBM, NCR, NSF, TRW, and UC MICRO.

The mechanism we present focuses on improving performance for I/O intensive applications performing no direct manipulation of the transferred data. However, this does not preclude the use of "stacked" kernel processing modules to perform common functions required by the applications, in a fashion similar to Streams [Rit84]. These processing modules may even be implemented in hardware where appropriate; for example, compression-decompression. Today's workstation technology is not capable of providing the high-bandwidth I/O demanded by applications like HDTV-quality video or even NTSC-quality video, when moved about uncompressed and processed by the CPU in real time. Regardless, the mechanism we present complements the functionality provided by the `read()` and `write()` system calls, and does not preclude their use in programs. However, for those applications not requiring the buffering interface supported by these calls, our mechanism can improve performance, and make possible I/O intensive applications using workstation technology available today.

Design Goals

Our goal in modifying the I/O system is to demonstrate improved data throughput and increased CPU availability in current systems, with asynchronous operation between devices or files without adversely affecting the standard UNIX I/O architecture and interface. An attractive strategy for achieving these goals is to decouple process execution from I/O data flow by introducing a new system call based on the following design principles:

- Avoid unnecessary data copying
- Provide asynchronous operation
- Support concurrent I/O operations

A new system call `splice()` achieves these goals with two unique features. First, the call has no buffer interface as do the UNIX `read()` and `write()` system calls because data is not moved to

and from user space. Although we could have opted for a shared-memory interface, as several others have suggested, we wished to avoid the memory interface entirely. Interfaces based upon a memory abstraction require data to be moved to and from user address space in quanta and format dictated by the user process, with additional constraints imposed by the local machine's virtual memory hardware. With splice, I/O data is beyond the reach of a process; it may never even go through the machine's main memory (e.g., in systems supporting peer-to-peer DMA). Consequently, the kernel can decide how to best transfer data between devices, given its knowledge of the source and destination transfer characteristics and device interfaces.

Second, the call operates asynchronously in a fashion similar to the asynchronous I/O calls present in several current versions of UNIX, and Windows NT [Cus93]. A calling process may continue user-mode execution while I/O is proceeding between objects. The process may regain control of the splice execution periodically by carefully adjusting the transfer size parameter (described below), but does not need to create threads or call specialized "async I/O" functions as several systems require.

An old-style telephone operator "patching together" two communicating parties is an appropriate analogy for the operation of splice. Splice may also be thought of as providing the "reverse" capability of the original Streams IPC pseudoterminal (PT) [Rit84] or the streams-based pipe implementation in 8th Edition UNIX [PrR85]. The PT in Ritchie's streams and pipes in 8th Edition provide IPC by cross-connecting file descriptors within the kernel. Splice, in contrast, provides the cross-connection of devices within the kernel as specified by the calling process.

Interface

Splice takes two UNIX file descriptors and an integer size as arguments. The file descriptors specify the source and sink of I/O data, respectively. They may refer to files, character device special files, or sockets. The size parameter specifies the number of bytes to be moved between the source and the sink; a special value indicates the splice should execute until an end of file condition is reached or the operation is interrupted by the caller. The splice operates asynchronously if either of the file descriptors have the FASYNC flag enabled, as set by a call to `fcntl()`. A calling program can opt to catch SIGIO to detect the completion of an asynchronous splice.

Example

The example in Figure 1 illustrates the use of splice in an application which plays back a digitized movie from a file.

The above code segment illustrates the use of splice to transfer a complete file (e.g., an audio file) as well as a partial file (e.g., frames of a video file). For audio, the splice moves digital audio samples asynchronously from the audio data file to the output DAC (D-to-A converter). The program assumes the audio DAC driver converts and delivers audio at the appropriate playback rate to match the recording rate in the file. Several audio device interfaces (e.g., Sun's `/dev/audio`) operate in this fashion. For

```
    ...
int audiofile, videofile;   /* digital audio/video files */
int audio_dev, video_dev;   /* output dacs */

audiofile = open("movie.audio", O_RDONLY);
videofile = open("movie.video", O_RDONLY);

audio_dev = open("/dev/speaker", O_WRONLY);
video_dev = open("/dev/video_dac", O_WRONLY);

fcntl(audiofile, F_SETFL, FASYNC); /* set async operation */

/* copy the audio information; return immediately */
splice(audiofile, audio_dev, SPLICE_EOF);

/* loop, delivering one frame every timer interval */
setitimer(ITIMER_REAL, &inter_frame_time);
do {
    rval = splice(videofile, video_dev, sizeof(video_frame));
    pause();    /* wait for timer to go off */
        /* it will reload automatically */
} while (rval > 0);

...
```

Figure 1: Playing back a movie

video, the program assumes a video device capable of displaying frames at a maximum rate faster than the recording rate of the source file. That is, a full-speed splice between the source file and the display device would play video too quickly to match the corresponding audio. Slowing the splice transfer rate is achieved by ensuring the FASYNC property is not set, and adjusting the size parameter to specify a limited transfer quantum (e.g., the size of a single frame for video). The calling process retains control of the transfer rate by making splice requests at appropriate intervals. A video "fast forward" or "slow motion" could be affected by adjusting the interval timer value. Moreover, splice requires no buffer handling by the user program, and provides support for multiple simultaneous I/O operations.

Implementation

Splice is currently implemented as a system call under Ultrix 4.2A, and has been tested on a DecStation 5000/200 and DecStation 5000/240. The code comprises about 3000 lines of C source code (including comments), and increases the kernel's object size by about 10% (to 1.9MB).

Background

The current implementation of splice supports file-to-file splices between files residing on local disk storage devices, socket-to-socket splices for the UDP transport protocol, and framebuffer-to-socket splices for sending graphical images and video. For brevity, this discussion outlines only the portions of the implementation relevant to the 4.2BSD-based filesystem. The splice implementation requires a buffer cache kernel interface, and makes use of the following buffer cache routines: `bmap()`, `bread()`, `getblk()`, `bawrite()`, `brelse()`, as well as the dynamic kernel memory allocator and callout list. This paper assumes basic familiarity with these functions. They are discussed in more detail in [LMK89].

Implementation and Operational Details

Assuming an entire file is to be copied, splice operates generally as follows. First the size of the source file is determined from information present in the gnode (Ultrix terminology for generic filesystem node). A special *splice descriptor* is dynamically allocated to keep state information about the data transfer. Placing all necessary information in this descriptor allows I/O to proceed without requiring the availability of the calling process' context. The entire list of all physical block numbers comprising the source file is determined by successive calls to `bmap()`. The list of physical blocks is stored in a dynamically allocated table in the splice descriptor. The destination file is mapped similarly to the source file, except a special version of `bmap()` is used for improved performance which avoids delayed-writes of freshly allocated, zero-filled blocks. At this point, all information necessary to proceed with an asynchronous data transfer has been stored in the splice descriptor, and user-mode execution of the calling process may be resumed.

Read-Side Operation

Data transfer between the source and destination files must be allowed to proceed without blocking; no guarantee can be made as to the availability of the calling process' context. New versions of the kernel routines `bread()` and `getblk()`, with the calls to `biowait()` removed, provide most of the needed functionality. The physical block number is retrieved by indexing into the table in the splice descriptor by the logical block number on the source file. A call to the new `bread()` will schedule a read request and return immediately, instead of blocking awaiting buffer completion in `biowait()`. A handler function is installed in the buffer preceding the call to the driver's strategy routine by setting the `B_CALL` bit and `b_iodone` fields in the buffer header. When a read completes, the read handler is invoked which in turn schedules a write by placing a reference to the write handler at the head of the system callout list.

Write-Side Operation

The write side of splice is called via the callout list with a locked buffer containing valid data just acquired from the source file. The callout list is used to decouple the I/O access periods at the source and destination I/O devices. Lock-step behavior is avoided by introducing the asynchrony provided by the callout list; this improves performance by allowing I/O operations at the source and destination points to proceed simultaneously.

New fields in the buffer header structure indicate the splice descriptor and logical block number which are associated with a buffer's data. Thus, several buffers may be in transit simultaneously and need not be maintained in sequential order. The logical block number, retrieved from the read-side buffer header, is used to index into the splice descriptor to determine the destination physical block number for the current buffer's data. The physical block number is used to request a buffer header using a modified version of `getblk()` which avoids allocating any real memory to the buffer, but rather only sets the `b_bcount` field in the new buffer header to the requested size. The data pointer in the new buffer header is saved and altered to point to the same address the data pointer in the read-side buffer does, so both buffers share a common data area. We thus avoid copying between cache buffers. The size and flags fields in the buffer header are also saved and updated to match the corresponding fields in the read-side buffer header.

At this point a write handler is installed in the header (by assigning the `b_iodone` in the buffer header), and an asynchronous write is performed by

calling `bawrite()`. The write handler begins execution after the asynchronous write has completed. It retrieves a pointer to the source-side buffer for the current logical block number from the buffer just written and frees it by calling `brelse()`. It then frees the buffer just written similarly. Finally, a read request restarts the entire cycle.

Flow Control

Flow control for splice cannot be achieved by causing the calling program to block; in any case, the caller is not directly responsible for initiating intermediate read or write requests, so causing it to block would provide little benefit. Instead, rate-based flow control based on the completion rate of write requests is employed. Each splice descriptor maintains a count of the number of pending read and write requests. If the number of pending reads and the number of pending writes drop below pre-specified watermarks (currently 3 and 5, respectively), the write handler will issue up to a pre-specified number (currently 5) of additional reads. These values must be set such that the source is not underutilized and the destination is not overwhelmed.

Experiments

We performed several experiments to measure the effectiveness of splice. The goal of these experiments is to demonstrate improvement in CPU availability and I/O system throughput. These improvements are achieved by reducing copying and switching overheads when using splice rather than using user-level read/write sytem calls to transfer data between files.

Configuration

We performed all experiments on a DecStation 5000/200 equipped with 32 MB memory using a 3.2 MB buffer cache. The DecStation 5000/200 MIPS R3000 processor is clocked at 25 Mhz and includes a 64 KByte instruction and 64KByte write-through data cache. Cached memory read throughput is 21 MB/s, uncached CPU read rate is 10 MB/s, and partial-page write throughput is 20 MB/s [DEC90].

We used Digital's RZ56 and RZ58 SCSI disks for performance measurements. The RZ56 provides an average rotational latency of 8.3 ms, average seek time of 16 ms, and a to/from media peak data transfer rate of 1.66 MB/s. The RZ58 provides an average rotational latency of 5.6 ms, average seek time of of under 12.5 ms, and to/from media peak data transfer rate of 3.1-3.9 MB/s. The RZ56 provides 64 KB of read-ahead cache, and the RZ58 provides 256 KB of read-ahead cache segmented into 4 read-ahead requests [DEC92].

The performance improvement of splice is most pronounced when applied to devices producing or consuming data at high rates relative to the CPU execution rate. To determine how splice would perform when using fast devices, we also implemented a RAM disk. The RAM disk is a device driver with a character-special and block-special device interface upon which a UNIX file system may be created. Consequently, we were able to compare splice's performance when using a fast device versus when using relatively slow devices (i.e., disks), all of which require execution of the same file system code. The RAM disk driver uses 16MB of statically allocated memory from the kernel's BSS region, leaving a free memory pool of 5MB.

CPU Availability Test

We accomplish the goals of measuring CPU availability and throughput by executing a CPU-bound test program in three different environments:

- **IDLE**: execution of the test program with no other programs running
- **CP**: execution of the test program concurrent with a process executing the UNIX program *cp*, copying a large regular file from a file system located on one physical disk to a file system on a different physical disk
- **SCP**: identical to CP, except a splice-based copy program *scp* is used rather than *cp*

Baseline performance indices are obtained by executing the test program in the IDLE environment and noting how long a fixed set of operations take to complete. To measure changes in CPU availability, we compare the amount of time required for the test program to complete the same number of operations in the CP and SCP environments. To measure device-to-device file I/O throughput, we ensured a read cache cold start condition by performing large file I/Os through the buffer cache before taking measurements. We ensured write-through behavior for the cache in the case of writes by using only asynchronous writes for SCP and calling `fsync()` on the destination file for CP. Many of CP's delayed-write blocks are forced to disk in any case because the file sizes tested are larger than the buffer cache size.

Disk Type	Reduction Due to CP	Reduction Due to SCP	Improvement SCP over CP
RAM	49.3%	81.4%	65.1%
RZ58	63.6%	84.2%	32.4%
RZ56	63.8%	79.8%	25.1%

Table 1: Improvement in CPU Availability (Copying 8 MB File)

Table 1 shows the relative performance degradation of a CPU-bound process when executing concurrently with a process copying an 8MB file using either *cp* or *scp* (i.e., the CP or SCP environments), with disks of various performance characteristics. We also performed tests with significantly larger file sizes, with results that were statistically indistinguishable from the 8MB representative case listed

above. Column one lists the type of disks being used, including the two SCSI disks described above and the RAM disk driver (recall that the RZ56 is slowest disk, and the RAMDISK is the fastest). Columns two and three show the percentage reduction in execution rate experienced by our test program in the CP and SCP environments, respectively, as compared to the IDLE environment. Thus, a percentage reduction of X% means the process's execution rate was a factor of X/100 of the "IDLE rate," which is the execution rate in the IDLE environment. For example, in the CP environment using RAMDISKs, the test program executed at 49.3% of the rate it would execute in the IDLE environment, thus running approximately twice as long in the CP environment. Finally, column four indicates the percentage improvement in execution rate when executing in a SCP environment compared to a CP environment; an improvement in execution rate of 100% is a doubling in execution speed. This improvement factor measures the relative number of additional CPU cycles available to run the test program when contending with splice-based rather than read/write-based I/O. For example, when using RAMDISKs, a program's execution rate will improve by 65.1%, effectively shortening the execution time by a factor of approximately 3/5 due to splice-based I/O (rather than using read/write), see Table 1.

When contending with *cp*, the test program executes between 1/2 and 2/3 of its speed without contention. However, when contending with *scp* which uses the CPU and memory more efficiently when doing I/O, the test program executes at 4/5 or more of its speed without contention. Thus, processes will experience a 25 to 65 percent improvement in execution speed when contending with splice-based I/O versus read/write-based I/O, depending on the device speeds. With faster devices, splice's effect on performance improvement becomes more dramatic.

Throughput Tests

We now consider throughput performance of splice-based I/O versus read/write-based I/O. Table 2 shows the achievable throughput using *scp* vs. *cp* when copying files. For the throughput tests, we disabled the test program used to produce Table 1, so the figures in Table 2 represent maximum attainable throughput measures assuming an otherwise idle CPU. Column one indicates the disk type, columns two and three represent the throughputs measured for copying an 8MB file using *scp* and *cp*, respectively. The fourth column indicates the percentage improvement in throughput of *scp* as compared to *cp*. Thus, splice-based copying can operate at just under 1.8 times the maximum throughput of read/write-based copying using fast devices (in this case, RAMDISKs). However, when using relatively slow devices such as today's SCSI disks, the disk transfer time dominates the overall throughput measurement and the benefit of splice is minor.

Discussion

The performance improvements achieved by splice result from two modifications to the I/O subsystem:

- shortening the path data must travel between devices by eliminating the need to move data to and from user space
- bypassing context switch overhead between the reading of the input device and writing to the output device, leaving flow control and timing of block transfers (within a single splice operation) to the kernel

Except for small modifications for non-blocking behavior, we have not made fundamental modifications to the buffering, scheduling, or block allocation strategies present in most UNIX systems. We plan to investigate these areas, as well as the performance of our SCSI device driver, with the expectation of higher performance.

Related Work

The work described here relates to the general problems of system overhead encountered with large throughput I/O loading. Dean and Armand [DeA92] explore the effect of microkernel-based operating system design on data movement performance, suggesting the desirability of including device manipulation code directly in user processes to avoid copying. Similar suggestions are made by Forin et. al in [FGB91], who suggest the mapping of device registers directly to user-level processes. Govindan and Anderson [GoA91] describe "memory-mapped streams" as a mechanism for moving continuous media data between address spaces using shared memory. These approaches require the interface of I/O devices to appear as memory objects, and must therefore be mappable to a process' address space.

Disk Type	SCP Throughput (MB/s)	CP Throughput (MB/s)	%-Improvement of SCP over CP
RAM	3.34	1.88	77.7%
RZ58	0.59	0.55	7.3%
RZ56	0.37	0.37	0.0%

Table 2: Mean Throughput Measurements (Copying 8 MB File)

Several architectures restrict the ability to map devices, especially to user address space. Furthermore, we believe the data transfer size granularity should be specified by the application, rather than being constrained by details of the VM hardware.

In an approach similar to ours, Pasieka et. al [PCM91] suggest the UNIX *ioctl* be used to pass handles between source and destination devices, referring to kernel-level data objects. Their scheme decouples data movement from the application but requires user process execution to effect a data transfer between devices.

Conclusions

This study suggests a viable and promising augmentation to the standard UNIX system interface for I/O intensive applications. The experimental results indicate a reduction in process-related overhead which contributes to improved performance, both in terms of throughput and CPU availability during I/O periods. The programmer interface is convenient for the class of applications wishing to move data unaltered from one device or file to another. We believe the class of I/O intensive applications to be a large one, including multimedia programs wishing to connect audio and video ''streams'' between devices and files.

Acknowledgements

The design and implementation strategy of splice evolved over many discussions between the authors and others at UCSD and Berkeley, especially Keith Muller and Kirk McKusick who contributed substantially with their knowledge of UNIX filesystem internals. We would also like to thank members of the program committee for their critical comments and pointers to related work.

References

[Cus93] H. Custer, **Inside Windows NT,** Microsoft Press, Redmond, WA, 1993.

[DEC90] Digital Equipment Corporation, "DECStation 5000/200 KN02 System Module Functional Specification, Rev 1.3", Workstation Systems Engineering, Palo Alto, CA, Aug, 1990.

[DEC92] Digital Equipment Corporation, "Information and Configuration Guide for Digital's Desktop Storage: Focus on the RZ Series of SCSI Disk Drives", Disk and Subsystems Group, Palo Alto, CA, Feb, 1992.

[DeA92] R. W. Dean and F. Armand, "Data Movement in Kernelized Systems", *USENIX Workshop on Microkernels and Other Kernel Architectures,* Seattle, WA, Apr, 1992, 243-261.

[FGB91] A. Forin, D. Golub and B. Bershad, "An I/O System for Mach 3.0", *Proc. Usenix Mach Symposium,* Monterey, CA, Nov, 1991, 163-176.

[GoA91] R. Govindan and D. P. Anderson, "Scheduling and IPC Mechanisms for Continuous Media", *Proc. 13th Symp. on Operating System Principles,* Pacific Grove, CA, Oct, 1991, 68-80.

[LMK89] S. J. Leffler, M. K. McKusick, M. J. Karels and J. S. Quarterman, **The Design and Implementation of the 4.3BSD UNIX Operating System,** Addison-Wesley Publishing Company, 1989.

[PCM91] M. Pasieka, P. Crumley, A. Marks and A. Infortuna, "Distributed Multimedia: How Can the Necessary Data Rates be Supported?", *Proc. Usenix Summer Conference,* Nashville, TN, June, 1991, 169-182.

[Pas92] J. Pasquale, "I/O System Design for Intensive Multimedia I/O", *Proc. IEEE Workshop on Workstation Operating Systems,* Key Biscayne, FL, April 1992.

[PrR85] D. Presotto and D. Ritchie, "Interprocess Communication in the 8th Edition Unix System", *Proc. Usenix Winter Conference,* Dec 1985, 309-316.

[Rit84] D. M. Ritchie, "A Stream Input-Output System", *ATT Bell Laboratories Technical Journal,* 63, 8 (October 1984), 1897-1910.

Author Information

Kevin Fall is a Ph.D. student at the University of California, San Diego. He received a B.A. degree in Computer Science from UC Berkeley in 1988, and remained at Berkeley working for the Computer Systems Research Group on the BSD UNIX project until 1989. He then worked in the area of network security for Project Athena at the Massachusetts Institute of Technology. He received a M.S. degree in Computer Science & Engineering from UC San Diego in 1991. Kevin also teaches programming and TCP/IP networking courses for the UCSD University Extension. His research interests include operating system I/O support for high-speed peripherals and networks. Reach him via e-mail at kfall@ucsd.edu.

Joseph Pasquale is an Assistant Professor of Computer Science and Engineering at the University of California, San Diego, where he co-directs the UCSD Computer Systems Laboratory. He is also a Senior Fellow at the San Diego Supercomputer Center. Dr. Pasquale received the B.S. and M.S. degrees in Computer Science from MIT in 1982, and the Ph.D. degree in Computer Science from U.C. Berkeley in 1988. He is a recipient of the NSF Presidential Young Investigator Award in 1989 and the IBM Faculty Development Award in 1991. His current research interests are in the areas of operat-

ing systems and networks, particularly the design, implementation, and performance evaluation of I/O system and network software to support distributed multimedia (especially digital video and audio) applications and I/O-intensive scientific applications. Dr. Pasquale is currently involved in the design of the Sequoia 2000 Network, which connects the University of California campuses, and whose goals are to support high throughput transmission of scientific data and to provide real-time support for collaborative distributed multimedia applications such as video conferencing. His electronic address is pasquale@ucsd.edu.

Reach the authors via U. S. mail at: Computer Systems Laboratory, Department of Computer Science and Engineering, University of California, San Diego, La Jolla, CA 92093-0114.

Fremont: A System for Discovering Network Characteristics and Problems

David C. M. Wood, Sean S. Coleman, & Michael F. Schwartz – University of Colorado

ABSTRACT

In this paper we present an architecture and prototype implementation for discovering key network characteristics, such as hosts, gateways, and topology. The Fremont system uses an extensible set of modules to discover information, based on a variety of different protocols and information sources, rather than a single network management protocol. This approach allows more complete and timely information to be discovered than, for example, using only one protocol, even one as capable as the Simple Network Management Protocol. The discovered information is time-stamped and recorded in a database. The contents of this database are cross-correlated to form an increasingly complete network picture, to direct further discovery, and to highlight inconsistent information.

Introduction

The Scenario

Everything looked OK on the network monitor when your boss walked in, complaining that she couldn't get to the Ancient History server in the Classics department. Now you're in trouble. Everything you normally monitor is obviously up, but the problem just won't go away. But no problem, if you have the tool that will tell you what the route is supposed to be to get to the Classics subnet. You had heard before that they were on the network, but you never knew that the connection was via a Sun workstation / gateway in the Athletics department. After a quick call, you can report back to your boss that the coach has plugged his workstation back in, and the history server should be accessible in ten minutes.

Well, probably the campus network wouldn't really depend on careful administration in the Athletics department. Nonetheless, tracking changes and problems in a campus network is difficult, because authority and responsibility for various network segments is distributed across multiple organizations. Even on the segments that *are* well controlled, users (particularly those departing the institution) have no incentive to report that they are removing their host from the network. It is usually not an emergency, but it is useful to find out about such activities, particularly before one runs out of network addresses on a segment.

Motivating The Approach

"What's the big deal?" you ask. "I can use traceroute[9] to track down this routing problem." Perhaps. But traceroute really works best when the network is functioning properly. When there are problems, traceroute alone may not identify the problem. There may be multiple paths between a host and destination; over time the routes may change. Maybe you are experiencing a performance bottleneck, rather than a network partition. Observing that traffic passes across a subnet with a large number of hosts attached to it may help explain the problem. Other problems may also arise. For example, on any large network occasionally two hosts get configured with the same IP address. This generally makes communications impossible for either host. Detecting this problem is relatively easy if you have a tool that remembers the IP and Ethernet associations longer than the usual timeout of the ARP cache[15].

"Well," you continue, "I can use traceroute to find the path, and then I can use the Simple Network Management Protocol (SNMP)[2] to check the packet arrival rates at each gateway along the way. That handles the performance problem. For multiple network addresses I would..."

Sure, you can do it manually, but this is the sort of thing that computers are supposed do well. The fragmented nature of the existing network management tools makes life difficult.

There are two types of problems with current network management tools. First, each tool takes a particular perspective, and cannot support network management functions that require other perspectives. For example, SNMP treats the network as a set of instrumented devices. It can only retrieve information from nodes where SNMP agents are running, and cannot perform active probing tasks (such as traceroute). Moreover, since it focuses on managing known devices, it cannot aid in the discovery of devices or services, and the network manager must invest significant effort in configuring these tools.

More generally, different sources of network information have different characteristics with respect to timeliness of discovered information, discovery expense, danger of generating network problems (such as broadcast storms), and completeness of discovered information.

These differences lead to the second problem: network managers must manually cross-correlate information obtained from several tools. The tediously detailed nature of this information makes it virtually impossible for a network manager to harvest all of the useful information that is potentially available. Instead, when problems arise the manager probes the small window of information needed to solve that problem.

What is needed is a framework for network management that combines the various tools into an integrated system. The system should observe many different aspects of network state, and integrate the information into a coherent picture. Given such a picture, the network manager can learn of problems earlier, and can view a current picture of various aspects of the network more easily.

System Description

Overview

John Charles Fremont was a nineteenth century explorer in Colorado, California, and other parts of the western United States[4]. Inspired by his versatility, we named our network discovery system after Fremont. The Fremont system architecture is illustrated in Figure 1. In this figure, light lines indicate control flow, and heavy lines indicate data flow.

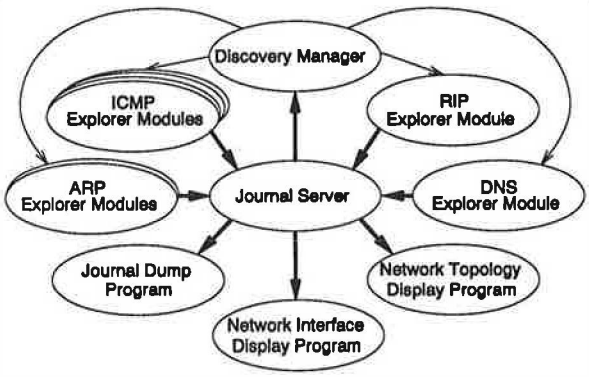


Figure 1: Fremont System Architecture

The Fremont system is based on an extensible suite of *Explorer Modules*, each of which uses a commonly available, existing network protocol or information source to uncover network information. This range of modules supports a broad set of discovery mechanisms and techniques. Some Explorer Modules actively probe the network, sending packets out into the network, and watching their effects. Other Explorer Modules generate no network traffic, and instead quietly observe the network activity around them. For example, passive packet monitoring allows routing information to be collected without imposing added processing load on a gateway. More active, directed probing allows both local and remote networks to be examined without the need for installing specialized network monitoring hardware or software.

Just as Fremont the explorer kept a dated journal of his activities, the Fremont system records discovered information in a central repository, which we call the *Journal*. This Journal is managed by the *Journal Server*, which serializes updates, timestamps and records the data, and answers queries from programs that wish to interrogate the Journal.

The activities of any good explorer are heavily influenced by experiences along the way. The Fremont system supports this function by way of a *Discovery Manager*. The Discovery Manager interrogates the Journal, and compares information discovered from the various Explorer Modules to determine a more complete picture of network characteristics (such as topology), and direct further discovery. Because every discovered feature is time stamped with its original date of discovery, its last change, and its last verification, network changes are easy to track. The Journal can also be interrogated by user interface agents, as will be discussed later in this paper.

The Fremont system is intended for a variety of network environments. Because all modules communicate via BSD sockets, there are no restrictions about the physical location of individual modules. Moreover, the system can be replicated at multiple sites, exploring different networks, and sharing information among the replicated components.

Explorer Modules

The current Fremont prototype supports 8 different Explorer Modules, based on 4 different information sources. For each information source, we give a brief description of the nature of the source, what type of information the Explorer Modules can discover using that source, and a detailed discussion of how each Explorer Modules operates. The detailed discussions include the conditions under which each information source can and cannot successfully discover network information, as well as how the information can be cross-correlated with other discovered information to provide insight into the represented networks.

The reader interested only in a high-level understanding of the Fremont system can read the first 2-3 paragraphs in each Explorer Module subsection, and skip the detailed discussions.

Address Resolution Protocol Explorer Modules

The Address Resolution Protocol (ARP) provides a mapping between Medium Access Control (MAC) and network layer addresses (e.g., between Ethernet and IP addresses)[15]. Whenever a host tries to send a packet to another host on a shared subnet, the sending host must first look up the Ethernet address in the local host's ARP table. If there is no entry for that IP address, the host must broadcast an ARP request for the destination Ethernet address.

The host that claims to have that IP address will reply to the requester.

Fremont has two Explorer Modules that discover and record the mappings provided by the Address Resolution Protocol. This information can be used in many cases to determine the manufacturer of the discovered interface[1]. It can also aid in determining gateways, locating changes in interface configurations, and discovering multiple interfaces with the same network layer address.

Fremont's ARPwatch Explorer Module passively monitors ARP message exchanges, and builds a table of Ethernet/IP address pairs for the directly attached subnets. Because this module uses the Network Interface Tap (NIT) feature of SunOS, this module must be run with system privileges.

Fremont also has an EtherHostProbe[12] Explorer Module, which attempts to send an IP packet to the UDP Echo port of each host in a range of addresses. Doing so causes the originating host to generate ARP requests, the responses for which are entered into the host's ARP table, and then read by the EtherHostProbe Explorer Module. For each address probed, one ARP request is broadcast. In addition, if there is a host on the network with the probed address, it will generate an ARP reply. The originating host will then send the UDP packet to the Echo port of the probed host, and the probed host will, if so configured, reply to that packet. In summary, there is an ARP request broadcast for each address probed, and then two or three additional packets will appear on the network for each responding host. The module limits the rate of generated packets to four per second. It does not use the Network Interface Tap and does not require special privileges.

Fremont has two ARP-based Explorer Modules because each module has different strengths and weaknesses. The ARPwatch module requires special privileges, and will not discover hosts that are not recipients of traffic from other hosts. This module generates no network traffic, and can be left to run for long periods of time. The EtherHostProbe module generates traffic, and does not require special privileges. It provides more thorough discovery, and will finish in an amount of time limited by the number of addresses probed.

Both modules share some common limitations. Both are limited to gathering information only about hosts that are on a directly attached, locally shared subnet (e.g., hosts on the same Ethernet as the one on which the Explorer Module is running). Both modules must ignore "proxy" ARP replies, where a gateway issues an ARP reply for hosts that are behind the gateway. This is easily done when the remote hosts are on a different subnet, but some gateway devices will reply for a set of addresses on the local subnet. Our solution in these cases is to recognize the device type when the multiple IP addresses are reported for a single Ethernet address. Except for this case, multiple IP addresses for a single Ethernet address usually indicates that a system has been reconfigured. Multiple Ethernet addresses for a single IP address usually indicates a misconfigured host, which is using the IP address assigned to some other host.

Internet Control Message Protocol Explorer Modules

The Internet Control Message Protocol (ICMP) provides a variety of network layer information, as part of the Internet Protocol (IP)[16]. Fremont has four Explorer Modules based on ICMP. Since ICMP packets are usually processed at high priority in router queues, the ICMP Explorer Modules take precautions not to send them too frequently.

The first two ICMP Explorer Modules are based on ICMP "Echo Request" and "Echo Reply" messages. These messages are used by the UNIX "ping" program, to test if a remote host is reachable[14]. The purpose of these two modules is to identify operational interfaces attached to the network. The first Explorer Module is based on sequential pings. This module sequences through a range of addresses, recording when it receives replies.

The second ICMP Explorer Module is based on broadcast pings. This module sends an ICMP Echo Request to the broadcast address of the subnet being probed. These *directed broadcasts* tend to be less successful than sequential pings on a subnet with many hosts, because closely spaced replies can cause many collisions. However, if used carefully, broadcast ping can be an effective interface discovery tool for large subnets of class A and B networks. In particular, it works well if the address space is large but there are not very many hosts on the individual subnets. In these cases, a sequential search of the address space would take a long time.

Directed broadcast packets that are broadcast with a time-to-live field larger than one can cause severe broadcast storms, due to incorrect networking software implementations or configurations in even just one host on a network. To minimize this problem, the broadcast ping Explorer Module sends packets with minimal time-to-live values (determined dynamically, in a fashion similar to the sequential increase mechanism used by traceroute; see below). To avoid broadcast storms, many gateways are configured not to broadcast packets that have a directed broadcast address as the destination address. This avoids the problem, but also reduces the effectiveness of the broadcast ping module.

[1] Throughout this paper we use the term *interface* to refer to a network interface, i.e., a separately addressable network connection to a machine.

The third ICMP Explorer Module is based on ICMP mask request/reply messages for determining the subnet mask of a network interface. This is not as widely implemented as the echo request/reply. In fact, some implementations allow the interface to be configured not to respond to subnet mask requests, to avoid problems with hosts configuring themselves based on incorrect subnet mask replies. Nonetheless, Fremont uses this feature of ICMP to discover and record the subnet masks of all the interfaces that it has already discovered. Fremont uses the collected subnet masts to aid in determining the network structure. It also uses the gathered information to detect conflicting subnet masks on different interfaces of a subnet.

The fourth ICMP Explorer Module uses a technique similar to Traceroute [9], to determine the route a packet would take from source to destination. Normally, when a packet is sent towards a destination, it has a "Time-To-Live" (TTL) field that is set large enough that the packet may reach its destination, yet small enough that undue network resources will not be consumed if the packet gets caught in a routing loop. As each router along the path receives the packet, it decrements the TTL field. If a router decrements the TTL field to zero, that router is supposed to drop the packet and send an ICMP Time Exceeded packet back to the source of the original packet. If the TTL is still non-zero, then the router forwards the packet on to the next router closer to the destination.

Traceroute takes advantage of this feature to determine routes by sending a sequence of UDP packets towards a destination host. The first packet in this sequence has a TTL of 1. That packet is dropped by the first router along the path, and that router sends an ICMP Time Exceeded message back to the source. The next packet is sent with a TTL of 2, and the second router along the path drops the packet and returns an ICMP Time Exceeded message. This process continues until the TTL of the packet sent is just large enough for the packet to reach its destination. The sequence of ICMP Time Exceeded messages returned to the source provides a trace of the routers along the path to the destination. The packets are sent to a port on the destination host that is unlikely to be used. Thus, when the packet arrives at the destination, it will typically cause the destination host to send either an ICMP Protocol Unreachable or ICMP Port Unreachable message.

Fremont's Traceroute Explorer Module uses this mechanism to determine the structure of the network surrounding the host on which the module is running. It does this by using the traceroute scheme to identify gateways and the subnets to which those gateways are connected.

Not all routers perform correctly as described above. Some hosts send their Unreachable message back to the source using the TTL field from the received packet, causing the packet not to arrive back at the source until the TTL of the original packet is large enough for an entire round trip. The Traceroute Explorer Module can handle most of the common failure modes, and hence provides an excellent means of discovering network topology.

The Traceroute Explorer Module sends packets towards three host addresses on the destination subnet, in an attempt to maximize the amount of information discovered. For example, if the module is sending packets towards subnet 128.138.238 (netmask of three bytes), then it sends packets to 128.138.238.0, 128.138.238.1, and 128.138.238.2. If a host receives a packet that is addressed to host zero on the subnet, the host is supposed to treat that packet as though it were addressed to that host. The module therefore sends to host zero on the destination subnet to maximize the chance of getting a reply from a node on that subnet. We send to two other addresses on that subnet, on the assumption that although one of those addresses may actually be the interface address of the gateway that accepted the packet addressed to host zero, the other address will not be that same gateway. The gateway should then send a final ICMP Time Exceeded message as it decrements the TTL to zero and drops the packet destined to some other address on the subnet.

The Traceroute module continues to send packets towards as yet unreached destinations while waiting to timeout packets it has sent to other destinations. It ensures that no more than eight packets per second appear on the network as a result of this parallel activity. With a ten second timeout for a response, this can result in up to 80 outstanding packets at any one time. However, most of these packets will have been lost somewhere, and the module will just be waiting for them to time out.

The Traceroute module is designed to operate on a local, high-speed network. Although the module will work across shared, low-speed serial connections, it stops tracing towards a particular destination if that trace reaches any of several national backbone networks.

Because it will receive ICMP Time Exceeded messages from only the single closest interface on the routers along the traced path, the Traceroute module will only discover half the interfaces traversed. Running this module from multiple locations in the network will acquire more complete information about the router interface addresses.

The current implementation of this module does not make any attempts to discover multiple paths, although the internal data structures are in place to accommodate this. Alternate routes can be discovered by running the module from different points in the network or, in some cases, simply by running it at different times. For example, if a lower priority, redundant path exists between two

locations, that path will be discovered only when the primary path is down. Since this module, like all of the other Explorer Modules, stores its information in the Journal, the Journal will contain more complete information aggregated from multiple invocations of this module.

Routing Information Protocol Explorer Module

The Routing Information Protocol (RIP) uses broadcast messages to advertise routes to particular networks, subnets, or hosts[7]. Although it has fairly limited capabilities, RIP is widely used. Each RIP packet from a router contains a list of network addresses and metrics. No subnet mask information is contained in these packets, so routes to networks, subnets, or hosts are determined by comparing the subnet mask of the receiving host to the address being advertised. Subnet advertisements are not propagated outside of the network to which the subnets belong.

Unfortunately, not all RIP sources are trustworthy. Many badly configured hosts "promiscuously" rebroadcast all learned routing information without regard to the subnet from which that information was learned. This gives the false impression that the host may really have a separate route to the advertised networks. Fremont's RIP Explorer Module attempts to identify those RIP sources that appear to be operating in this erroneous manner.

The RIP module monitors RIP advertisements on shared subnets, building a list of hosts, subnets, and networks as they are seen in the advertisements. The collected data is recorded in the Journal, and used as clues for further discovery probes.

Like the ARPwatch module, the RIPwatch module uses the Sun NIT with a packet filter to watch the RIP packets on the shared subnets. This means that this module must run with system privileges, and that the module can see no further than the directly attached subnets.

Domain Naming System Explorer Module

The Domain Naming System (DNS) stores name, address, name server, and other information about interfaces in a distributed, hierarchical database[13]. Names and addresses are both stored in two distributed tree structures. One tree is organized to permit easy address lookups given a domain name. The other tree is organized to permit easy name lookups given an IP address. This latter tree is often called the "reverse" domain.

Fremont's DNS Explorer Module searches the appropriate subtree for all addresses in a specified network. The primary purpose of this module is to discover network topology by identifying gateways. This module was derived from the "nslookup" program, which is part of the Berkeley Internet Name Domain server distribution[10]. Nslookup understands how to format queries for all of the different types of data that the DNS supports, and how to interpret the results.

The DNS module retrieves the set of all address-to-name mappings from a domain, using "zone transfers". It does this by descending recursively into the DNS tree starting from a specific point, in a manner similar to Ganatra's Census program[6]. This technique creates no more network or name server load than is caused by a secondary DNS server.

The DNS module also uses ICMP Mask Requests to retrieve the subnet mask from one of the first hosts discovered on the desired network.[2] This is usually one of the name servers, thus increasing the likelihood that the returned mask is correct. Using the subnet mask and the information obtained from the DNS tree, the module tries to determine which sets of interfaces comprise gateways. It does this by looking for several different matches. The most obvious case is when multiple IP addresses correspond to the same machine name. The DNS module also looks for multiple names for the same address, and then looks for matches within those groups. It further looks for names which differ only by "-gw" or similar naming conventions.[3] This module also looks for "designated" gateways[13], but this does not appear to be a widely implemented use of the DNS.

The DNS module records in the Journal the number of hosts on each subnet and the highest and lowest addresses assigned on each subnet. Since the module has the complete set of name/address pairs for the network being examined, it could send all of this information to the Journal. However, because this information is readily available from the DNS, we do not record a name/address pair if it is the only information that we have involving an interface.

Journal

Each Explorer Module collects some subset of the available network data, depending on where the module runs and which protocol or information source it uses for discovery. This information is recorded in the Journal.

Most recorded data are used to provide a representation of the network. These data may then be used to answer user queries about the network entities and structure. In addition, some of the recorded data are used as a guide to further

[2]The DNS module was written before we had a Journal server. Since it needed the subnet mask in order to know how to allocate interfaces to subnets, and since we wanted to make this as automatic as possible, we chose to have the DNS module invoke the Subnet Mask module.

[3]In the future we will identify *possible* gateways using other weaker heuristics, tagging the resulting entries in the database with a "questionable quality" flag.

discovery. For example, the data collected from RIP packets provide strong indications about the existence of specific other networks and subnets. This information is used by the traceroute Explorer Module to improve its performance.

The Journal data are grouped into records representing interfaces, gateways, and subnets. Table 1 shows the primary fields that are maintained for interface records.

MAC layer address
Network layer address
DNS name
Subnet mask
Gateway to which this interface belongs

Table 1: Interface Fields

Gateways are represented as collections of interfaces. For gateways, we also record the subnets to which they are connected. The reason for doing this is that the Traceroute Explorer Module is able, in some cases, to determine the subnet to which a gateway is attached without being able to determine the address of the interface on that subnet.

For each discovered subnet, we record a list of gateways attached to that subnet. Note that there are cases where we may have discovered a subnet, but do not yet know what gateways are connected to that subnet.

All data items are stored with the date and time of initial discovery, last change, and last verification. This information is useful for observing several network characteristics. For example, we can see when hosts have been removed from the network. When this happens, Fremont stops updating the interface data record (except perhaps via the DNS Explorer Module). A network manager can observe this, and then contact the owner of the missing host to verify that the network address can be reused.

Because it is the shared place where observations are stored, and because there are several Explorer Modules recording complimentary findings there, the Journal is more that just the sum of its parts. For example, the fact that the same Ethernet address is observed by two ARP modules running on different subnets is not significant until that information is written into the Journal. Only then, because of the common storage, can that gateway be discovered. Similarly, both the Traceroute and DNS Explorer Modules collect information about gateways, and store that information in the Journal. Because the two modules use different techniques, the resulting data in the Journal are more complete than might be determined by either module acting alone.

Journal Server

The Journal Server maintains an in-memory representation of the Journal data, which it writes to disk periodically and at termination.

As noted in the Journal description, the stored data are grouped into records representing each interface, gateway, or subnet. Each of record is stored in a linked list for that type of data. The lists are ordered by time of last modification, so that the most recently changed items are at the end of the list. The data records for interfaces are indexed by three AVL trees, for lookups by Ethernet address, IP address, and DNS name. This allows quick access to individual data records, as well as access to ranges of records. An AVL tree is also used to index subnet records by subnet address. Gateways can be accessed by any one of their interfaces. The storage requirements are shown in Table 2.

Record	Bytes/Record
Interface	200
Gateway	84
Subnet	76

Table 2: Journal Storage Requirements

Note that in a distributed environment, no one Journal Server would need to store information about much more than the local network. Hence, storage requirements are modest. For example, a 25% full class B network (16k interfaces) with 192 subnets used (and an equal number of gateways) would require under four megabytes of memory.

The Journal Server responds to three primary requests: Store/Update, Get, and Delete. These requests are supported through a common library of access and data transfer routines that the Explorer Modules, Discovery Manager, and data analysis and presentation programs use. The Get function may return multiple data records depending on the selection criteria in the request.

Discovery Manager

The purpose of the Discovery Manager is to decide what information needs to be collected and what Explorer Modules should be invoked to collect those data. The Discovery Manager initializes itself by reading a startup/history file containing the address of the Journal Server, and the command name, invocation frequency, and information about recent runs for each Explorer Module.[4] It then opens a connection to the Journal Server and retrieves the data related to the attached subnets. It next adjusts the schedule for running any particular Explorer Module, based on the data already collected. The

[4]The startup/history file was implemented before we had a Journal server. In the future we will move this data to the Journal.

startup/history file records what each Explorer Module needs for input, and what features it discovers. The current list is shown in Table 3.

As the Discovery Manager runs the various Explorer Modules, it updates the startup/history file, which is used to determine what modules to run next. For example, if the Discovery Manager sees that 20 of 400 interfaces recorded in the Journal do not have subnet masks recorded and that this was true before the "subnet mask" module was last invoked, then the Discovery Manager will not shorten the interval until the next invocation of that module. This ensures that the resulting exploration effort is as fruitful as possible.

Info. Source	Module Name	Inputs	Outputs
ARP	ARP-watch	none	Enet. & IP addr. matches (over time)
	Ether-HostProbe	IP addrs.	Enet. & IP addr. matches (immediately)
ICMP	Sequential-Ping	IP addr. range	Intf. IP addrs.
	Broadcast-Ping	Subnets or Nets	Intf. IP addrs.
	Subnet-Masks	IP addrs.	Subnet Masks
	Traceroute	Subnets, Nets, or nothing	Intfs. per gateway; gateway-subnet links
RIP	RIP-watch	none	Subnets, Nets, Hosts
DNS	DNS	Network number	Intfs. per gateway

Table 3: Explorer Module Input/Output

When the Discovery Manager starts an Explorer Module, the Discovery Manager has several mechanisms for directing the Explorer Module. The particular mechanism for each Explorer Module is recorded in the Discovery Manager startup/history file. Most Explorer Modules, if given no specific direction, will examine the directly connected networks or subnets.

Presentation Programs

The ultimate purpose of Fremont is to provide some insight into the network being explored. To this end, we have built three programs for viewing the data available in the Journal. The first program simply lists all of the data in the Journal. We used this for early debugging.

The second program presents the interface data in three levels of detail, using X window displays. The first level lists all interfaces in a particular network, including the network layer address, DNS name, and time since last verification of existence (ignoring time of last DNS verification). This gives an easy indication of when the interface was last observed on the network. The second level lists all subnet interfaces, including the MAC layer address (if available), an indication of whether or not this is a source of RIP packets, and an indication of whether this is one interface of a gateway. The third level lists all of the data items stored in the Journal for a particular interface. This program is useful for looking at the state of the network interfaces over time. With it, a network manager can note which machines are out of service.

The third program provides an X window display of the network structure, as represented in the Journal. This is built from the gateway and subnet records stored in the Journal. The program retrieves the network and gateway entries from the Journal, and dumps the data in the format expected by SunNet Manager[24]. We then use SunNet Manager to display the data, as illustrated in Figure 2 (showing the output for a part of the University of Colorado network discovered by Fremont). This use of Fremont provides a significant improvement to SunNet Manager. While SunNet Manager provides a *discover tool* that checks the routing table on the local machine to find subnets, it does not uncover the relationships between the subnets. Using SunNet Manager, the user must enter and maintain network relationship information manually. Fremont supports this function automatically.

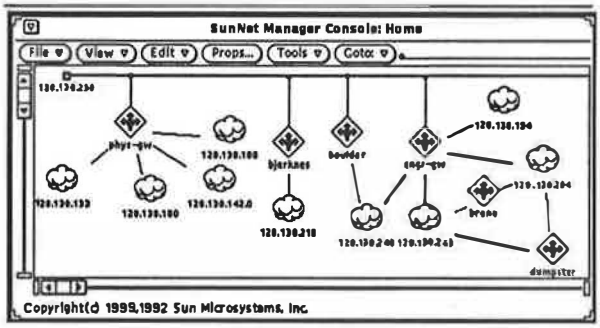


Figure 2: Discovering Subnets

Analysis Programs

We have two programs that analyze the Journal data to uncover possible network problems. The first lists subnet mask conflicts for all of the interfaces in the same network. With this information we can identify hosts that are not configured properly for a subnetted environment.

The second analysis program lists the possible conflicts between MAC layer and network layer addresses. In particular, we locate cases where multiple interface records have the same network layer address for different media access addresses, or vice versa. The first case represents either changing hardware or two different hosts using the same

network layer address. The reverse situation may represent a system configuration change, a gateway doing proxy ARP, or the multiple interfaces of a gateway.

Evaluation And Experiences

In this section we present measurements and comparisons of the various Explorer Modules, to help evaluate the overall cost and effectiveness of the Fremont system.

Network and System Load

Table 4 shows the intervals that we have found appropriate for network discovery, the time required for completion of each invocation, and estimates of the network and module host system load resulting from that module.

Using the intervals specified, network and system load is kept reasonable. We have also installed precautions to ensure that Fremont will not adversely affect the network on which it is running. For example, the system stops tracing towards a particular destination if it detects a routing loop. Also, the modules that use parallel network activity to improve performance limit the rate at which packets are generated. The modules that use the Network Interface Tap to watch an attached subnet minimize the load on the host system by packet filtering.

Discovery Effectiveness

Table 5 shows the results of a brief run of Fremont, exploring one of the subnets in the Computer Science Department at the University of Colorado. For this run, all active modules were run once. Results for the one passive-monitoring module (ARPwatch) are given after the first 30 minutes of monitoring, as well as after 24 hours. As can be seen, quite a few interfaces were discovered almost immediately, and monitoring network traffic for a day caused most interfaces to be discovered.

To compute the "% of Total" column, we presume that the DNS data are an accurate reflection of the network. In the case of the network we tested this is a reasonable assumption, because the people who operate that subnet are very conscientious about keeping the DNS current. In fact, when we scrutinized the DNS records, we found only two entries for which there were no real machines connected to the network. From this we concluded that the DNS data showed slightly more interfaces than actually existed. We did not find any interfaces on the subnet that were not in the DNS.

Table 6 shows measurements of discovering the subnets of the campus network. We have assigned about 114 subnets, but several of those are not in use at this time. The RIPwatch module discovered 111 subnets. This can be treated as the exact number of subnets since, if we cannot find a route to a subnet on campus, then effectively it is not connected to the campus network. The DNS module found 93

Module	Min/Max Interval	Time to Complete	Network Load	System Load
ARPwatch	2 hours; 1 week	continuous	none	minimal
EtherHostProbe	1 day; 1 week	1 sec/address	1 - 4 pkts/sec	minimal
SeqPing	2 days; 2 weeks	2 sec/address	.5 pkts/sec	minimal
BrdcstPing	1 week; 4 weeks	30 sec/subnet	short storm	short high load
SubnetMasks	1 day; 1 week	2 sec/address	.5 pkts/sec	minimal
Traceroute	2 days; 2 weeks	5 - 20 minutes	4 - 8 pkts/sec	moderate
RIPWATCH	2 hours; 1 week	2 minutes	none	minimal
DNS	2 days; 2 weeks	1 - 5 minutes	10 pkts/sec	high

Table 4: Explorer Module Characteristics

Module	Interfaces	% of Total	Reason for loss
ARPwatch	34	61	Run for 30 min
	50	89	Run for 24 hours
EtherHostProbe	48	86	Not all hosts up when run
BrdcstPing	42	75	Collisions
SeqPing	38	70	Not all hosts up when run
DNS	56	100	Not necessarily current

Table 5: Discovering Interfaces on a Subnet Results from 1 Run of Each Active Module

subnets. This is because not all subnet managers enter their interface addresses into the name service. The DNS module further found 31 gateways connecting 48 of those subnets. Note that each of Tables 5 and 6 showed only the modules relevant to their discovery task (interfaces for Table 5, subnets for Table 6). Not all modules are used for both tasks.

Module	Subnets	% of Total	Comments
Traceroute	86	77	Gateway software Problems
RIPwatch	111	100	Nearly all subnets advertized
DNS	93	84	Not all hosts name served
DNS	48	43	Subnets with gateways identified

Table 6: Discovering Subnets Results from 1 Run of Each Active Module

Observations

In the following paragraphs, we offer observations from our experiences with the various Explorer Modules. In particular, we address such features as reliability, completeness, time to completion, and network and system resource consumption.

The Sequential Ping Explorer Module is the simplest and most reliable of the modules, because virtually every host implements the ICMP Echo Request/Reply protocol. The load presented to the network is low, because request packets are sent only once every two seconds. This will result in one reply packet for each existing host. If the module receives no response to a packet after issuing one request to each destination address, it sends one more request packet to each destination that did not respond. The second request rarely succeeds on a local network unless either the network or the remote host are heavily loaded. Running this on a class C network takes between 9 and 18 minutes. Running this on an entire class B network address space would take between 36 and 72 hours.

The Broadcast Ping Explorer Module presents a brief flood of ICMP Echo Reply packets. On a network with many hosts, this can provide a stress test of collision handling implementations, and usually results in lost packets, including both ICMP Echo Replies and normal traffic. Therefore, the reliability of this module suffers. The tradeoff is that this module completes in 20 seconds on a directly attached network.

On our networks, the ARPwatch and RIPwatch Explorer Modules consume minimal system resources on the hosts running those modules. Neither module generates any traffic, and both use the NIT to reduce the resource demands on the machine running those modules. Similarly, the EtherHostProbe and Subnet Mask modules offer only minor loads to the network and the machine running those modules. Like the Sequential Ping Explorer Module, these modules can take a long time to examine a large address space.

The Domain Name System Explorer Module operates in two phases. During the first phase, the module makes DNS requests of a name server for the network being examined. The network load is noticeable while the module does "zone transfers", as required to descend the DNS tree below the desired network. This activity takes about half of the time used by this module. During the second phase, the module searches the collected information for gateways. This is CPU intensive, particularly during the search for names with suffixes indicating possible gateways.

The Traceroute Explorer Module is modest in the demands that it places on either the network or the host system. This is mainly because we understood that traceroute activity might have significant impact on the network. We therefore were careful to impose limits on the load presented by this module. This conservative approach expands the time that it takes for this module to complete its exploration. We recommend that this module only be used to explore high speed networks (Ethernets or faster). The module has command line parameters that allow it to be slowed down more than the default value, as should normally be done when used on a slower network.

We did not implement an SNMP module in the current prototype because SNMP was running on only a few machines in our local internet when we started this project. Furthermore, SNMP requires knowledge of community names, which limits its ease of use. We plan to implement an SNMP Explorer Module (see the Future Work section).

In the DNS Explorer Module, we looked for *distinguished gateways*, as described in RFC 1035[13]. While this information could be useful for discerning network topology, we found that it is rarely supplied in the deployed DNS databases. Many other types of information are similarly unavailable or incorrect, such as lists of Well Known Services (WKS) and host and operating system type information.

Why is some information (like host names, addresses, and mail exchange records) available and reasonably up-to-date in the DNS, while other data is notoriously bad? The answer is that the data that *must* be correct in the Domain Naming System for

proper operation in a networked environment generally *will* be correct. If a host system can function on the network without some particular piece of information being correct, current, or complete in the DNS, then it is quite likely that this information will be none of these. For example, the fact that a particular Well Known Service is running on a machine is more directly available in the distributed collection of /etc/inetd.conf files (which provide a list of the program locations for each service that is actually available on each machine). This is precisely why the WKS field in the DNS was deprecated in RFC 1123[1]. Network service information can also be determined by attempting to connect to a service, in the case of virtual-circuit based services[19]. Because of this, systems administrators rarely keep the WKS entries in the DNS up to date. These observations indicate that a name service works best for managing data needed for correct network operation, and that other types of data are better provided by a dynamic discovery process.

Related Work

A number of network management tools have been built using existing protocols[11, 17, 23]. However, these tools use only one or two sources of such information, and do not cross-correlate the data as Fremont does. Multiple information sources and existing protocols have been used to support resource discovery in other contexts as well, including Netfind (which discovers Internet user directory information)[20] and archie (which discovers files available via anonymous FTP)[5].

Robertson has built a system called *netdig*[18] that can discover network topology using SNMP. Several commercial network management stations also provide this capability. However, as with any use of SNMP, it is necessary to know the community name for every router in the network being examined. Most manufacturers' SNMP network management stations also offer some simple tools for drawing networks. The xnetdb program[3] does this at minimal cost, but it does no topology discovery, beyond connecting together hosts and gateways on the same subnet.

Future Work

We are currently extending Fremont to provide support for large internets, by caching data and supporting predicate-based queries to limit exchanged data to the parts that are needed. As a first step, we are making our software freely available, and encouraging people to set up Journal Servers throughout the Internet.

Another set of extensions in progress is adding Explorer Modules to use the two other protocols in their explorations. The first is SNMP. Although using SNMP requires knowledge of community strings, it is popular and powerful enough to allow improved topology discovery (as done by Columbia's *netdig* system). The second is Cisco Systems' Gateway Discovery Protocol (GDP). While not widely deployed, supporting GDP would help fill in some of Fremont's discovery gaps. A "promiscuous" mode network traffic monitor would be able to discover all communicating machines in a network. We will use this to extend our system into the discovery of network services.

We are also expanding our work with existing protocols. For example, beyond monitoring RIP advertisements, we plan to use directed probes to discover routing information, via the RIP Request and RIP Poll queries. The major advantage of doing so is that these requests and replies can be routed through a network, thus providing access to routing information on subnets other than just the local subnet. A problem, however, is that not all routers use RIP or respond properly to RIP Request or RIP Poll queries. Nevertheless, we expect to be able to identify some routers, and even some alternate paths using RIP queries [8].

Another area of Future work involves running the Traceroute Explorer Module from multiple points in a network. This is easy to do manually now, but will require remote execution capabilities in the Discovery Manager. We also plan to use "loose source routing", to look for multiple paths in the network. This feature of IP can allow the module to specify an intermediate router through which the traced path will be routed. We also plan to branch further from the local network, while continuing to minimize network impact. For example, if the network to be traced is only reachable through node G, and if G is exactly and always (for the duration of the traceroute run) H hops away from the host running the Traceroute module, then all traces can start with a TTL of H+1 rather than 1, because every packet will follow the same path from for the first H hops, and there is no need to continually re-trace the initial H hop path.

The data recorded in the Journal need to incorporate a more flexible notion of information *quality*. Currently we treat information discovered by some protocols as being of *better quality* than that discovered by other protocols. For example, data gathered using the ARP protocol are generally timely and correct, whereas DNS data are older and often subject to data entry errors. Thus, if the only indication of the existence of an interface is its record in the DNS, we would not add it to the Journal unless it appears to be part of a gateway. Similarly, we would like to have a flag to prevent continually retrying discovery of some datum that we know is unavailable. This would be similar to the negative caching concept that has been suggested for the DNS, in which the absence of an entry in the DNS could be locally cached to avoid unnecessary expense of future failed queries.

We plan further to examine the feasibility of extending the discovery processes to other protocols, particularly DECnet and OSI.

Our initial user inquiry agents focus primarily on rudimentary debugging tools with few graphical capabilities. We would like to have several tools that could provide real-time observation of the explorations and the discovered information, and a graphical visualization of the structure of the network as it is discovered. In the future, a more sophisticated interface could be integrated, perhaps from one of the commercial network management packages.

A final area of future work involves extending Fremont's graphical display mechanism to support dynamic updates, as new information is discovered.

Summary

The complexity of modern data communication networks has led to a situation in which network administrators must use a number of different tools to track changes and uncover problems in their networks. Because of no one tool provides a complete picture of the network, network managers have been forced to cross-correlate information obtained from several tools manually, often losing important information among the details.

The Fremont system provides a framework for network management that can combine many different tools into an integrated system. This approach represents an extension to the traditional network management paradigm, which treats the network only as a collection of instrumented devices (as in the case of SNMP). In addition to this paradigm, our approach supports passive traffic monitoring, active network probes, and information gleaned by cross-correlating data discovered from multiple sources. Because of this cross-correlation, Fremont provides more complete and useful information than any single network management system. It can also flag potential network problems based on inconsistencies in the discovered data. Fremont performs these functions without undue consumption of network or host system resources.

Table 7 summarizes the network characteristics that the current prototype discovers, based on the 8 different Explorer Modules we have implemented. This information is sufficient to provide detailed network maps, including topology maps (as illustrated in Figure 2), and tables showing the names and addresses of each host on each network, the local gateways used by each host, etc. In the future we expect to add route discovery capabilities to Fremont, at which time routing maps could also be produced.

Interfaces	Ethernet Address IP Address Name Subnet Mask Gateway Membership
Gateways	Interfaces on GW Subnets connected (topology)
Subnets	Gateways on Subnet (topology)

Table 7: Characteristics Discovered by Prototype

Table 8 summarizes the network problems that Fremont uncovers. The uncovered information can help network administrators solve a number of problems, such as those discussed in the Introduction section of this paper.

IP Addresses No Longer in Use
Hardware Changes
Inconsistent Network Masks
Duplicate Address Assignments
Promiscuous RIP Hosts

Table 8: Problems Uncovered by Prototype

In summary, the Fremont system provides an integrated framework for assisting a network manager in discovering network characteristics and trouble-shooting network problems. Because it makes use of many different information sources and network management tools, Fremont can form a more complete network picture than any one tool.

Prototype Software Availability

The Fremont software is available by anonymous FTP from ftp.cs.colorado.edu, in the directory pub/cs/distribs/fremont.

Acknowledgements

This material is based upon work supported in part by the National Science Foundation under grant NCR-9105372, and a grant from Sun Microsystems' Collaborative Research Program.

We thank Barb Dyker, Darren Hardy, Susan Smith, and the USENIX program committee for their helpful comments on this paper.

The Fremont system is based in part on the preliminary architecture described in[21]. This work is part of the Networked Resource Discovery Project at the University of Colorado[22].

References

1. R. Braden, "Requirements for Internet Hosts – Application and Support," Internet Request For Comments 1123, October 1989.

2. J. D. Case, M. Fedor, M. L. Schoffstall & C. Davin, "Simple Network Management Protocol," Internet Request For Comments 1157, May 1990.
3. H. Clark, *Xnetdb Software*, Ohio State University, June 1990. Available by anonymous FTP from thor.oar.net: /pub/xnetdb
4. F. Egan, *Fremont, Explorer for a Restless Nation*, Doubleday, Garden City, New York, 1977.
5. A. Emtage & P. Deutsch, "Archie – An Electronic Directory Service for the Internet," *Proceedings of the USENIX Winter Conference*, pp. 93-110, San Francisco, California, January 1992.
6. N. K. Ganatra, *Census Software*, Department of Computer Science, University of California, Santa Cruz, June 1992. Available by anonymous FTP from ftp.cse.ucsc.edu: pub/csl/census.tar.Z
7. C. Hedrick, "Routing Information Protocol," Internet Request For Comments 1058, Rutgers University, June 1988.
8. J. C. Honig, *RIPQUERY Software*, Cornell Theory Center, Cornell University, August 1991. Available as part of the gated distribution, by anonymous FTP from gated.cornell.edu: pub/gated/gated-2.1.tar.Z.
9. V. Jacobsen, *Traceroute Software*, Lawrence Berkeley Laboratories, December 1988. Available by anonymous FTP from ftp.ee.lbl.gov: pub/traceroute.tar.Z
10. M. Karels & J. Wood, *Nslookup Software*, University of California, Berkeley, September 1990. Available as part of the BIND distribution, by anonymous FTP from okeeffe.cs.berkeley.edu: /4.3/bind.4.8.3.tar.Z
11. K. Kislitzin, "Network Monitoring by Scripts," *Lisa IV*, October 1990.
12. P. E. McKenney, *EtherHostProbe Software*, SRI International, July 1988. Available by anonymous FTP from phloem.uoregon.edu: /pub/src/etherhostprobe/etherhostprobe.tar.Z
13. P. Mockapetris, "Domain Names – Implementation and Specification," Internet Request For Comments 1035, University of Southern California Information Sciences Institute, November 1987.
14. M. Muuss, *Ping Software*, U. S. Army Ballistic Research Laboratory, December 1983. Available by anonymous FTP from uunet.uu.net: /bsd_sources/src/ping
15. D. C. Plummer, "An Ethernet Address Resolution Protocol – Or – Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware," Internet Request For Comments 826, November 1982.
16. J. Postel, "Internet Control Message Protocol," Internet Request For Comments 792, University of Southern California Information Sciences Institute, September 1981.
17. W. C. Reissig, "Dynamic Network Management Using the Simple Network Management Protocol (SNMP)," Technical Report 90-08-04, Computer Science Department, University of Washington, Seattle, Washington, 1990. M.S. Thesis
18. S. Robertson, *Netdig Software*, Center for Telecommunications Research, Columbia University, August 1991. Available by anonymous FTP from ftp.ctr.columbia.edu: /pub/net/netdig.3.5.shar.Z
19. M. F. Schwartz, "A Measurement Study of Changes in Service-Level Reachability in the Global TCP/IP Internet: Goals, Experimental Design, Implementation, and Policy Considerations," Internet Request For Comments 1273, November 1991.
20. M. F. Schwartz & P. G. Tsirigotis, "Experience with a Semantically Cognizant Internet White Pages Directory Tool," *Journal of Internetworking: Research and Experience*, vol. 2, no. 1, pp. 23-50, March 1991.
21. M. F. Schwartz, D. H. Goldstein, R. K. Neves & D. C. M. Wood, "An Architecture for Discovering and Visualizing Characteristics of Large Internets," Technical Report CU-CS-520-91, Department of Computer Science, University of Colorado, Boulder, Colorado, February 1991.
22. M. F. Schwartz, "Internet Resource Discovery at the University of Colorado," To appear, *IEEE Computer Magazine*, Revised October 1992.
23. "FYI on a Network Management Tool Catalog: Tools for Monitoring and Debugging TCP/IP Internets and Interconnected Devices," Internet Request For Comments 1147, SPARTA, Inc., April 1990.
24. *SunNet Manager 1.1 Installation and User's Guide*, Sun Microsystems, Inc., 1991.

Author Information

David Wood holds a B.S. in Electrical Engineering from the Massachusetts Institute of Technology and a M.S. in Electrical Engineering from the University of Colorado. He is the manager of wide-area and campus-wide networking at the University of Colorado. He is also a Ph.D. student at the University of Colorado. He can be reached at Computing and Network Services, University of Colorado, 3645 Marine Street, Boulder, CO 80309-0455, or via electronic mail at dcmwood@spot.colorado.edu.

Michael Schwartz received his PhD in Computer Science from the University of Washington.

He is currently an Assistant Professor of Computer Science at the University of Colorado - Boulder. His research focuses on issues raised by international networks and distributed systems, with particular focus on resource discovery and network measurement. Schwartz chairs an Internet Research Task Force Research Group on Resource Discovery and Directory Service, and is on the editorial boards for *IEEE/ACM Transactions on Networking* and for *Internet Society News*. He can be reached at the Computer Science Department, University of Colorado, Boulder, CO 80309-0430, or via electronic mail at schwartz@cs.colorado.edu.

Sean Coleman received his B.S. in Engineering Physics at the University of Colorado. He is currently an M.S. student in Computer Science at the University of Colorado. He is also a system administrator for a network of Suns. He can be reached at the Computer Science Department, University of Colorado, Boulder, CO 80309-0430, or electronically at coleman@cs.colorado.edu.

The Enterprise Distributed White-pages Service

C. Mic Bowman & Chanda Dharap – Penn. State University

ABSTRACT

This paper describes the Enterprise user directory system. Enterprise is unique among directory services in three ways. First, clients identify people using an unordered set of attributes called a descriptive name. Descriptive names are easier to use and remember than, for example, X.500 distinguished names. Second, Enterprise supports an efficient distributed search facility. Rather than query every server in the system, Enterprise maintains several distributed indices that trim the set of potential servers to a fraction of the total. Finally, Enterprise provides a facility for automatically maintaining its database of information using existing repositories of information such as the `ruserd` daemon and the Sun NIS database. This removes the burden placed on users and system administrators to maintain the information. Enterprise is implemented as a collection of translators, resolution functions, and generators within a Univers descriptive name server.

1 Introduction

An internet community is composed of many different autonomous organizations. Each organization maintains a collection of information repositories that are exported to other internet participants. Often, these repositories include information about people. Enterprise is a distributed user directory service that helps clients retrieve information about people and organizations from a collection of autonomously maintained repositories. Enterprise supports searches through independent servers maintained at many sites around an internet. An Enterprise server consists of a database that stores information about people and organizations, query processing functions that access the information in the database, information generators that collect data from other repositories, and a communication package that supports access through several different client programs. Figure 1 shows the structure of an Enterprise server.

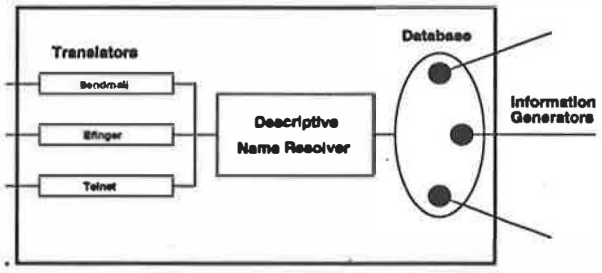


Figure 1: The internal structure of an Enterprise server

Enterprise is distinct among user directory services for three reasons. First, Enterprise allows clients to construct queries – also called *descriptive names* – using attributes that describe the requested object. For example, if a client attempts to resolve the descriptive name:

```
((name "Paterno")
    (research-interests "operating systems"))
```

Enterprise will return information about people named "Paterno" who are interested in operating systems. The actual set of objects returned depends on the resolution function the client selects. Resolution functions determine the scope of a query and the relative importance of the attributes. For example, the resolution function `equal` searches for people in the local Enterprise server that are described by every attribute in a descriptive name. If the client applies `equal` to the name given above, then Enterprise might return the following set of objects:

```
(( (phone "837-8319")
   (organization "University Park")
   (organization "Computer Science")
   (name "Joe Paterno")
   (research-interests "operating systems")
   (research-interests "communication")
   (uid "182cb0411.401d94"))
 ( (phone 367-5037)
  (organization "University Park")
  (organization "Computer Science")
  (name "Alice Paterno")
  (research-interests "operating systems")
  (uid "182cb0411.42f714"))
)
```

A different resolution function, `probe/lookup`, searches a collection of Enterprise servers and returns objects that match some of the specified attributes. `Probe/lookup` prefers objects that match important attributes such `name` and `organization`. Note that there is no arbitrary order imposed on the attributes that can be specified as part of a query. Thus, there is no distinguished name that must be remembered as with services such as X.500 [8]. However, to support rapid lookup for known

objects, Enterprise assigns a unique identifier to each object. A `uid` attribute describes the repository and location of information about an object. The combination of unique identifier and descriptive naming facilitates both white-pages and yellow-pages queries in Enterprise.

Second, Enterprise provides a global infrastructure for organizing naturally distributed collections of information for efficient resolution of descriptive names. The natural distribution of information in a collection of Enterprise servers reflects the autonomy of organizations in an internet: each repository in an internet contains information pertinent to the organization that maintains it. Each information repository defines a context for resolving descriptive names and is implemented within an Enterprise server. A collection of Enterprise servers forms a loose confederation and is bound together by a set of protocols for gathering and caching information about people, organizations and other servers. For example, an organization might designate a server as primary and ensure that it caches information about all other servers within the organization. In this case, an Enterprise client can find a list of available servers by contacting the primary server. In general, by actively maintaining caches, Enterprise creates several distributed indices for resolving descriptive names with relatively few messages. Compared to Enterprise, other directory services use more network resources, limit autonomy by centralizing information, or rely on users to maintain the information. For example, the X.500 [8] `SEARCH` facility provides rudimentary descriptive name resolution, but requires an exhaustive search of all information repositories. In a large internet the cost of this search is prohibitively high.

Finally, portions of the information contained in an Enterprise server are automatically collected using a facility called *information generators*. The information maintained in an Enterprise server may become out-of-date. For example, people change jobs and offices. New students are admitted to a University and others graduate. Existing directory services – such as the NIC name server WHOIS [14] – rely on users to maintain pertinent information. The weakness of this approach is that users are not reliable.

An Enterprise server collects information from a set of existing repositories; e.g., the `rstatd` and `ruserd` daemons, and the Sun NIS database. Since some information contained in these repositories is dynamic it is necessary that Enterprise periodically validate it. A generator serves as an interface between the Enterprise database and an authoritative information repository. It collects new information and validates existing information. In this way, the information in an Enterprise database is maintained indirectly through the maintenance of existing information repositories; the database is properly viewed as a cache of information from the authoritative repositories.

This remainder of this paper expounds upon these three important characteristics and describes some details of the Enterprise implementation. Section 2 describes the conceptual details of a single Enterprise server including the database, resolution functions, and generators it provides. Section 3 describes techniques for efficiently locating information within a distributed collection of Enterprise servers. Sections 4 and 5 describe the implementation and performance of a single Enterprise server and several client programs.

2 Local Information Management

The Enterprise user directory service consists of a collection of autonomously administered servers. Each server consists of a database of resource information, a suite of generators for maintaining the database, and several resolution functions that extract resource information from the database. The design of an Enterprise server is affected by two goals that are often in conflict: provide the local administrator with maximum flexibility to determine what information is stored in the database, and provide efficient query resolution within a large collection of servers. Enterprise attempts to balance these goals by establishing a small set of well-known mechanisms and policies for information management and functionality. This section describes the structure required by a single Enterprise server.

The Database

An Enterprise server maintains a lightweight relational database – also called a context – that contains information about people and organizations. Each external resource is represented by an Enterprise object that consists of a set of property/value pairs called attributes. For example, the following set of attributes represents a person in the Penn State Computer Science Department:

```
( (name "Joe Paterno")
  (organization "Computer Science Department")
  (account "joepa")
  (office "310 Whitmore Lab")
  (phone "865-5555")
  (research-interests "operating systems")
  (public-directory "/home/curly/joepa/pub"))
```

Generators add (register) and remove (unregister) attributes from objects in the database.

Since each organization is responsible for its own database, it is reasonable to assume that there will be variations in the representation of attributes and objects. For example, one database may register phone number attributes as a "phone" property similar to the example above. Another organization may separate the phone number attribute into home and office numbers. This flexibility creates a *vocabulary* problem. One solution to the vocabulary problem is to mandate a set of properties that must exist for

each object. Only queries that contain these attributes can be resolved. This approach limits the autonomy of organizations and forces organizations to agree upon a set of common properties. The vocabulary problem can also be solved by eliminating properties altogether, thus eliminating the need for clients to know the names of properties. The attributes are flattened into a single property that contains unstructured keywords. This approach has the advantage that clients do not need to know the property names in order to query a server. However, some semantic information is lost. For example, there is no way to distinguish between an office and home phone number. As will be shown in Section "Global Infrastructure", this semantic information is the key to efficiently resolving queries among a large set of servers.

Enterprise solves the vocabulary problem by combining important characteristics of each approach. The attributes registered for an object are broken into three classes: *mandatory*, *recommended*, and *optional*. Mandatory attributes are those that must be registered for an object; e.g., a person must have attributes for the `name`, `organization`, and `account` properties. The set of mandatory attributes is the same for each Enterprise server and includes properties most often sought by clients, such as a user's name. Recommended attributes are special in that Enterprise maintains structures to facilitate efficient resolution of queries that involve recommended attributes. However, recommended attributes need not be registered for every object. For example, Section "Global Infrastructure" describes a distributed index involving zip codes and city/state pairs. Recommended attributes for a user include `zip-code`, `phone-number`, `city`, `state`, and `interests`. Optional attributes can be added by participating organizations as required. For example, the `public-directory` property gives information about where files may be left for a person. Each Enterprise server defines a special property, called `optional`, that is the union of all optional attributes registered for an object. In this way, a client can specify a particular optional attribute if the property is known, or a keyword for the `optional` attribute if the property is not known.

A *type object* is a special object that describes the mandatory, recommended, and optional attributes for different classes of resources. For example, the person type is minimally defined by the following type object:

```
( (name "Person")
  (mandatory "name" "organization" "account")
  (recommended "zip-code" "phone-number"
               "city" "state" "interests")
  (optional "public-directory"))
```

Each organization can specialize the set of optional attributes for a particular resource class. Recommended attributes can be removed from the base type by any organization. However, new recommended attributes can be added only when a corresponding distributed search index is created. The set of mandatory attributes is immutable. Type objects are stored in the database and have mandatory attributes `name`, `mandatory`, `recommended`, and `optional`. A client that needs to know the properties supported by a particular server can look at the type objects defined within that server's database. Note that the type of an object is often included as an optional attribute. Enterprise defines types for users, organizations, types, resolution functions, and servers.

Generators

The reliability of the Enterprise directory service depends on the quality of information within the system. If a server's database contains out-of-date and inaccurate information or if some important information is missing, then the results of a query cannot be trusted. Since the attributes of an object can change – people change phone numbers, students graduate, research interests change – the information in an Enterprise database must be validated occasionally. An information generator is a mechanism

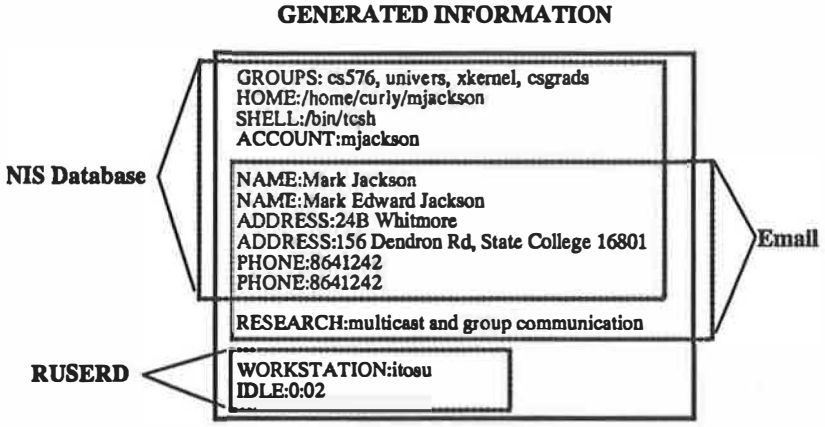


Figure 2: Enterprise mapping protocols

that queries an authoritative information source whenever attributes in the database need to be validated.

A generator is defined by two protocols: a mapping protocol and an agreement protocol. Intuitively, a mapping protocol maps information from an external repository into the Enterprise database. If we consider the information in a repository as defining a relation, then a mapping protocol maps attributes in the repository's relation into the relation defined by the Enterprise database. Operationally, the mapping protocol is implemented as a function that collects information from a repository, converts it into the Enterprise internal format, and constructs an index so that the information can be searched efficiently. Note that since authoritative information repositories within an organization differ greatly, there is no standard set of mapping protocols. Rather, Enterprise supports the development of site specific mapping protocols, each customized for a particular repository. The most common mapping protocols are implemented as clients of an existing service.

Figure 2 shows the attributes collected by three mapping protocols that we have found to be useful for our university environment. The three mapping protocols are as follows:

- `sendmail`: send mail to graduate students at the beginning of each semester requesting any modifications to currently registered attributes.
- `nis-database`: use information in the Sun NIS database or `/etc/passwd` file to generate objects for new accounts and for accounts that have changed.
- `usage`: use information from the *rusersd* daemon to determine the machine a person is currently using.

An agreement protocol defines a method for handling conflicts between similar attributes collected from different repositories. That is, the agreement protocol selects an appropriate value for a property given that several repositories maintain pertinent information. For example, Figure 2 shows that several attributes can be generated by more than one mapping protocol; e.g., address information potentially exists in both the NIS database and the information supplied by the user. The agreement protocol considers information gathered directly from the user as the most reliable. In general, when a conflict occurs, user supplied information is assumed to be accurate and the other values are discarded. Note that the agreement protocol recognizes that some properties can have several valid values such as the two address attributes that describe the user in Figure 2.

Enterprise collects information from authoritative repositories when the information in the database becomes out of date. Since properties maintained by generators can change at different rates – e.g., most student information can be gathered once a semester, but the processor a user is currently using may change hourly – Enterprise associates a time to live with each value in the database. When the server receives a query, it checks the time-to-live field of attributes in the database. If any are out of date, the corresponding generator is invoked. When the generator returns the query is processed. In this way Enterprise performs lazy updates. While there are other methods for generating information, for instance triggered updates and polling [10], lazy updates have the advantage of limiting the overhead of maintaining information at the cost of an occasional delay in query processing.

Resolution Functions

Enterprise supports both white pages queries where the client specifies a unique handle for an object, and yellow page queries where the clients specifies a potentially ambiguous description of an object. To support white pages queries, each object in an Enterprise database is given a unique identifier that remains consistent over the lifetime of the object. A client can retrieve information about a person or organization by dereferencing the unique identifier.

To support yellow pages queries, a client submits a descriptive name consisting of a list of attributes, and the name of a resolution function to an Enterprise server. The resolution function selects objects from the database that are, in some way, represented by the attributes in the name. In an ideal environment, it would be sufficient to have a single resolution function – i.e., the `select` operation from relational databases – that returns any object for which all attributes in the name are registered. However, given that some attributes provide more accurate responses, some are more diligently maintained, and some are trusted more by the client, it is necessary to provide a way for specifying certain preferences about how descriptive names are to be resolved. Consider, for example, a client who wants information about a person named Joe Paterno at Penn State who probably works in the Computer Science department. The client prefers information about a person named Joe Paterno in the Penn State Computer Engineering department instead of information about a Joe Paterno in the Virginia Tech Computer Science department. In this case, the `name` and `university` attributes are required; the client is unwilling to accept any object that is not described by these attributes. However, the `department` attribute is optional in the sense that the client is willing to accept objects that are described by conflicting attributes.

Enterprise defines a suite of resolution functions with respect to a preference hierarchy [2]. These functions include the following:

- `equal` Return those objects that precisely match all specified attributes. This function

expresses the belief that the descriptive name contains equally preferred attributes and that the information in the database is trusted. This function corresponds to the relational `select` operation.

- `all` Return those objects that precisely match at least one specified attribute. The returned objects are ordered by the number of matching attributes. This function is useful for browsing through the database when the client does not trust the information she possesses.
- `lookup` Return those objects that match attributes defined as `mandatory` in a type in preference to those that match `recommended` attributes; return objects that match `recommended` attributes in preference those that match `optional` attributes. This resolution function expresses the belief that the classification of an attribute gives some hints about the quality of information in a database. For example, queries containing attributes for zip code and phone number – both `recommended` properties – can be resolved more easily and more accurately than names consisting of attributes for a user's public directory – an `optional` property. This is because most servers will not maintain information about `optional` properties.
- `custom` The `custom` resolution function is based on the yellow pages function for processors described in [12]. The attributes in the query are partitioned into two sets: a set of optional attributes and a set of required attributes. The resolution function returns objects that match all required attributes and as many optional attributes as possible. This function allows the client to explicitly state the validity of certain attributes.

The set of resolution functions defined by an Enterprise server is not fixed, but it must contain these four functions. Most servers will define other resolution functions as well; e.g., functions specialized for a certain type of object like `finduser` or `findorganization`. Like types, an object is stored in the Enterprise database for each resolution function defined in a server. A client can determine the resolution functions provided by a server by searching for function objects.

3 Global Infrastructure

The previous section focused on the role of resolution functions in accessing information within a single Enterprise server. However, since the Enterprise user directory is distributed across many servers, a facility is required for locating servers that contain interesting information. Thus, finding information about a person or organization is a two step process. First, the descriptive name is mapped into a set of candidate servers. The set of candidates is determined according to some semantic mapping between properties of objects and properties of servers. For example, consider a descriptive name that contains the attributes `(organization "Penn State University")` and `(organization "Computer Science Department")`. It is obvious that the best place to look for objects that match these attributes is in the server maintained by the Penn State Computer Science Department. A search involving the attributes given above should be limited to the Penn State Computer Science department server. Since information in the Internet tends to be distributed based on organizational and geographical orientation, most semantic relationships map properties of people to an organization or a geographical region. The second step in resolving a name is to use the resolution function to select from each candidate server those objects that match the attributes specified in the name. Enterprise defines a function called `probe` that performs these two steps. The function `probe` takes the name of a resolution function and a list of attributes as arguments, and returns the objects computed by applying the resolution function to the attributes in one or more servers.

Limiting the set of servers that `probe` must search is the key to efficient query resolution. To accomplish this, Enterprise servers maintain a cache of information about other servers in the system. That is, each Enterprise server contains in its database several objects that represent other servers. The attributes in a server object describe the geographical location of the server, the organization that is responsible for maintaining it, and the protocol used to access it. We say that a server $S sub 1$ *knows-about* a server $S sub 2$, if it contains an object that represents $S sub 2$. The *knows-about* relation defines the Enterprise name space. It can be viewed as a network where network nodes and links represent Enterprise servers and communication channels. From the network perspective the problem of finding candidate servers is the problem of routing a query packet through the network defined by the *knows-about* relation. By manipulating the name space – e.g., adding new edges by caching server descriptions – Enterprise maintains a global infrastructure for limiting the scope of query resolution. The remainder of this section describes three techniques that Enterprise uses.

Backbone Routing

A general method for locating a set of servers creates a topology within the knows-about relation similar to the topology of the Internet. The backbone of the knows-about network consists of a small set of well-known servers that cache information about many other servers. The information in a well-known server is updated using two different generators. The first generator maps information about conventional Enterprise servers into the

database of the well-known server. When a new server is added to the system it advertises itself to one or more well-known servers by sending attributes that describe its geographical location, administrative organization, and access protocols. After being added to the system, a server periodically sends its information to the well-known servers. Rather than validate existing values, the object is deleted from the well-known server when its attributes time-to-live expires. Thus, as servers are removed from the system, the corresponding object is removed from the well-known server.

The second generator is responsible for maintaining connections between well-known servers. In order to guarantee that the knows-about network is not partitioned, the well known servers must actively update information about one another. The generator responsible for maintaining this information is similar to the one used for conventional Enterprise servers. As for conventional servers, when a well-known server is created and periodically after it is created, it advertises itself to other well-known servers. However, this generator is different in that the advertisement is sent to *all* other well-known servers. Another difference is that the readvertise rate and time-to-live are much shorter for well-known servers, forcing updates to occur more frequently. Note that when a well-known server advertises itself to other servers, it may include information it possesses about conventional servers. In this way routing information can be disseminated among the servers that comprise the backbone.

Finding a set of candidate servers is very similar to routing a packet through the Internet: send the packet to a gateway connected to the backbone, let the gateway forward it to other nodes within the backbone, and finally deliver it to the appropriate destinations. In this case, the address is a set of attributes that describe geographical or organizational properties of "interesting" servers. Note that a resolution function is forwarded along with the descriptive name. When the query reaches a destination, the resolution function is applied to the attributes to select objects from the database. The originator of the query collects responses from the various sites and returns the union to the client. Note that this approach is similar to the WAIS directory of servers where one or more servers maintains information about many other servers [9].

Hierarchical Routing

To resolve queries that contain geographical attributes such as zip code, city, state, or phone number, Enterprise uses a hierarchical topology maintained within the name space. Consider the topology defined by viewing the digits of a zip code as fields in a network address. This topology divides the name space into several primary regions defined by the 10 possible first digits of the zip code. Each primary region is divided into ten subregions for the second field, and so on. A server maintains connections with at least one other server in each level of the hierarchy. Intuitively, a server contains more detailed information about servers with a longer common prefix, i.e., those servers closer to it in the address space. With this topology, every server is the root of a tree that spans all servers in the system. For example, consider the connections for server 3.2.3 in Figure 3. This server connects to two servers – 1.3.2 and 2.2.2 – that differ in the first field of the address, two servers – 3.1.2 and 3.3.1 – that match the first field but differ in the second field, and two servers – 3.2.1 and 3.2.2 – that match the first two fields but differ in the third. To resolve a query with this topology, a server searches its database for the server with the largest common prefix and sends the query to that server. This moves the query closer to its destination. The new server will have more detailed information about the next step in the path. For five digit zip codes, the query will reach an appropriate set of servers using less than five messages.

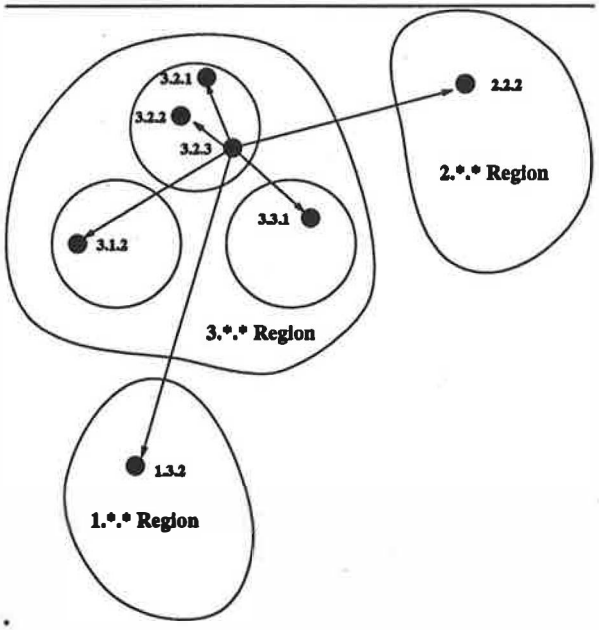


Figure 3: Hierarchical connections maintained by server 3.2.3

Special Interest Routing

Properties such as `interests` are less specific and less uniform than geographic properties so techniques such as hierarchical routing are far more difficult to implement. A better approach is to model the natural development of special interest groups and cliques. People tend to associate with people who have similar interests. Within the Enterprise name space, servers develop close associations with other servers that have similar interests. For example, a server that contains people working in operating systems would cache objects that represent other servers that contain operating systems

researchers. To find people interested operating systems, a client finds one server with operating system researchers and follows the pointers to other servers with useful information. Since the number of special interest groups is fairly small relative to the number of objects in the system, it is possible to designate certain servers as special interest repositories. These servers form their own special interest group. To facilitate location of special interest groups, the "clique of cliques" servers advertise themselves to other servers in the system. Thus a server maintains pointers to servers in its own special interest group and to one or more servers in the "clique of cliques" group. Figure 4 shows an example name space.

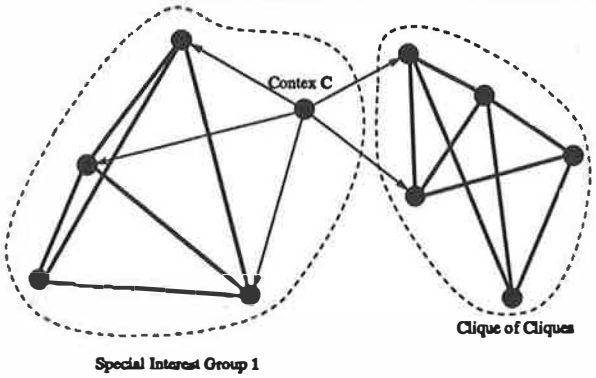


Figure 4: Clique connections of server C

There are two important tools used to maintain the special interest groups: *questors* and *advertisers*. A questor is a low priority broadcast query that progresses through the system collecting information on behalf of a special interest group. Unlike a normal query, a questor does not come from a client. Rather, a questor is originated by a special interest group and returns results to a set of servers – specifically those servers that are part of the special interest group. Its primary purpose is to build the name space. An advertiser is a low priority function call that is passed from server to server. When an advertiser arrives at a server, it deposits some information in a public area and notifies the system administrator. The system administrator examines the new information and decides whether to add it to the permanent information already in the server. An advertiser is useful for announcing new services.

4 Implementation

Each Enterprise server is implemented on top of a Univers descriptive name server [3]. A Univers name server can be customized via selection of storage mechanisms, translators, and generators. The server consists of three major components: an access manager, a database manager, and a program interpreter. The access manager provides a communication framework for accessing the server. It allows the definition of transport channels for various kinds of data handling. A client accesses the server through one of the supported transport channels. Each channel has an associated translator that converts input into the server's internal format and output into the format defined by the communication channel. Presently, Enterprise defines three transport channels: one for TCP clients such as telnet, one for the `finger` protocol [18], and one for electronic mail messages.

The database manager exports operations that access and maintain the information within the server's database. The database is partitioned into one or more *contexts* based on administrative domains. The database manager exports operations for selecting objects from a context, adding or removing objects from a context, and updating the attributes that describe an object.

The program interpreter translates a client's query into operations that access the database and supports a programmable interface for defining new constructs that extend the functionality of the server. This interface, realized by a Scheme-based language [17], allows clients to define new resolution functions. The access manager and database manager are entirely implemented in C. The program interpreter is implemented as a combination of Scheme and C code. The foundation for the interpreter is a public domain, C-based implementation of Scheme called Xscheme [1] that supports byte-compilation of Scheme programs. In order to provide the necessary functionality and to improve efficiency, Univers adds new primitives and modifies existing ones.

Generators

As mentioned earlier, a generator is defined by two protocols that actively maintain the information in the server's database: the mapping protocol that communicates with the external repository and the agreement protocol that merges gathered information with existing information. The Univers generator interface supplies the following functions to support the attachment of generators to portions of the database.

- `attach-generator` Associates an external daemon with the Univers generator. This function identifies an external repository from which information will be collected.
- `register-tag` Registers a group of properties with a generator. This specifies a mapping from properties defined in an external repository to attributes maintained in the server.
- `generate` Defines a conversion routine for translating data gathered from an external repository into attributes that can be registered for objects in the database.

Enterprise defines a `usage` generator to maintain information about users currently logged onto a host as shown in Figure 5. The generator interface

is approximately 400 lines of C code. The following properties are registered with the `usage` generator: `hostname`, `account`, `logged-on-from` and `login-time`. The core of the `usage` generator is the `usaged` daemon which collects information from `rusersd` daemons on machines across several networks. `Usaged` uses the interface provided by Sun RPC. When triggered, the daemon broadcasts an RPC call to the `rusersd` daemon. The information collected is sent back to the Enterprise server. The `usage` generator caches information in a private store. When new information is added to the cache, a flag is set in every property registered with the generator indicating availability of new data. Whenever there is a query for information maintained by any registered properties, the generator is checked for new information if the available flag is set, and triggered to collect information if the associated time-to-live time has expired. The `usage` generator uses account names as a key in mapping attributes from the `usaged` daemon to server objects.

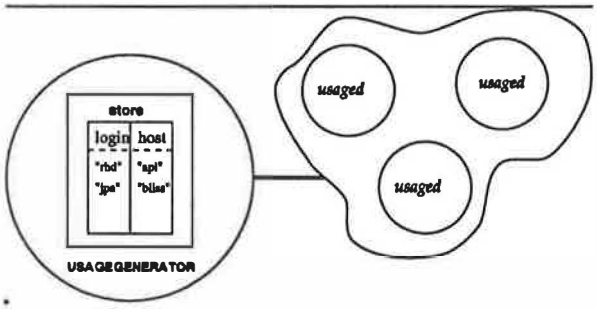


Figure 5: The usage generator and its attached tags and daemons.

Note that the `usage` generator provides unique attributes for login information; the information it collects is not collected by any other generator. Thus, there is no agreement protocol defined for it. This is not the case with the `sendmail` generator. The `sendmail` generator sends an electronic mail message to everyone in the organization. The message contains a template for the user to fill out and send back. The generator collects information from messages that are returned and adds it to the database. In the Penn State Computer Science department the `sendmail` generator is triggered once per semester. Since some information collected by the `sendmail` generator is also collected by the NIS generator, an agreement protocol is necessary. The agreement protocol prefers information from the `sendmail` generator about personal properties such as name, phone number, and interests, but not account name or home directory. When the `sendmail` generator updates the database the information is stored in a separate store called the priority store. To resolve a query, the priority store is searched first. If the information in the priority store fails to identify a user or organization – because, for example, information from users who failed to respond to the `sendmail` generator is missing – then the standard database is checked. In this way, the agreement protocol gives preference to information that comes directly from the user. i.

Clients

Enterprise supports access through three different transport channels: a raw TCP channel, a finger channel, and an email channel. Each channel is associated with a translator.

The TCP translator accepts raw data from clients such as telnet, and passes it on to the Univers program interpreter. Two special keywords are defined by the TCP translator: `:connect` and `:disconnect`. The translator reads a single expression from the channel. If the expression is the keyword `:connect`, then it continues to process expressions until a corresponding `:disconnect` is received. Otherwise, a single expression is read and evaluated, and then the channel is closed.

The Enterprise finger translator converts TCP finger packets into expressions that can be evaluated by the server. The finger translator recognizes two types of queries: queries from the standard finger client and queries from the Enterprise enhanced finger client. Queries from the standard finger specify a single keyword. The finger translator converts this into a query using the `equal` resolution function on name and account properties. Queries from the enhanced finger include attributes and a resolution function and are processed accordingly. The following example demonstrates the *efinger* interface (broken to two lines for display purposes):

```
efinger "name:N Joy,state:Pennsylvania,
        department:Computer Science"
```

If a resolution function is not explicitly provided, `efinger` uses `probe/equal`. The translator formats the selected objects appropriately and sends the result back to the client.

Unlike the above two translators, the email translator involves several additional manipulations. An alias `enterprise-query` is set up to receive electronic mail. Messages are archived in a file as they arrive. A perl script – run as a cron job – locks the archived file, connects to the email translator and sends the body of one message at a time. The header is parsed to obtain sender address information [5]. The message body is converted into an expression and processed by the program interpreter. The results are then packaged into a message and a reply is sent to the sender via sendmail. Currently the message body can contain one call to an existing resolution function. Email queries are particularly useful for resolution functions like `all` that often generate large replies.

5 Performance

Currently, Enterprise maintains data for five departments at the Pennsylvania State University. The data gathered from these departments is used for testing only. We incorporated information from the NIS databases and "/etc/passwd" files from the following five departments: Computer Science, Mathematics, Statistics, Chemistry and Astronomy. Enterprise is currently running on Sun SPARCstations. The servers for this test were run on two SPARCstation SLCs in the Computer Science department at the Pennsylvania State University.

Table 1 describes the data used the test environment. The first column specifies the administrative organization that supplies the information; e.g., the "cmpsc" database is maintained for the Department of Computer Science. The second column identifies the server on which the database resides. The total number of objects in a database and the size of the checkpoint files provide some perspective on the size of the experiment. Note that approximately 28K of data in a checkpoint file is Univers specific information. The actual space used to store object attributes is about 48 bytes per attribute. This is about a two to one increase in the size of the data.[1]

The average number of attributes per object indicates the number of properties maintained by the respective administrative domain. Some of the information from the NIS database was removed at the department's request for privacy reasons. For example fields like `home-directory` and `login-shell` were often removed. This is a clear example of an administrative organization specializing the contents of a server to accommodate certain operational constraints.

Times for resolving queries within a single server are shown in Table 2. The test involved selecting objects from the database that match a randomly chosen set of attributes. We also tested the performance of remote resolution. A query generated by the efinger client which must access both servers is resolved in about 300 milliseconds. This compares favorably to the time necessary to resolve finger queries using the standard `fingerd` daemon. Note that the total time spent by in efinger ranges between 0.9s to 1.5s depending on the number of users that match the given name. The largest amount of time is spent displaying the results on the client's screen.

[1] It is possible to achieve significantly better performance by using Univers storages mechanisms optimized for lookup times. However, the number of bytes required to store an attribute increases dramatically.

Function	Iterations	Attributes	Millisecs
probe/all	500	1	14
probe/all	500	2	20
probe/all	500	5	22
probe/equal	500	1	13.9
probe/equal	500	2	20
probe/equal	500	5	21

Table 2: Performance of single server lookups.

6 Related Work

Enterprise is an outgrowth of the Profile naming service developed at the University of Arizona [13]. Profile uses unstructured attributes to access a database of information constructed from "/etc/passwd" files. It supports a notion of preferences and its resolution functions are closely related to the required Enterprise resolution functions. However, Profile does not support automatic searching through a collection of servers and does not support automatic validation of information.

Both Enterprise and the OSI/CCITT X.500 user directory service [7] [8] facilitate identification of users and organizations in a distributed environment. X.500 supports white pages services using a distinguished name that consists of an ordered set of attributes. However, unlike Enterprise the attributes in an X.500 distinguished name do not necessarily describe properties of the named object. Rather, the attributes identify a path through a global hierarchy called the Directory Information Tree (DIT). X.500 supports distributed resolution of yellow-pages queries through the SEARCH facility. However, the performance of SEARCH is limited because X.500 provides no auxiliary data structures to constrain the scope of the operation. Thus, SEARCH broadcasts a query to all servers within a particular DIT subtree.

Database	Server	No. Objects	Data Size	Avg. Attributes
cmpsc	apl.cs.psu.edu	519	348K	14
math	apl.cs.psu.edu	246	190K	11
stat	apl.cs.psu.edu	113	119K	11
chem	bliss.cs.psu.edu	164	119K	11
astro	bliss.cs.psu.edu	179	119K	8

Table 1: Test Data Size

Several projects such as Indie [6] and Nomenclator [11] have proposed schemes for augmenting the X.500 search facility to improve its performance. X.500 does not specify a policy for maintaining information within a Directory System Agent.

Two name servers, CSNET [16] and the NIC [14], provide centralized descriptive user directory services for Internet participants. The CSNET name server defines a name resolution function in which a given set of attributes are partitioned into *mandatory* and *optional* subsets. Names are resolved giving preference to precise matches of the mandatory attributes. The NIC name server (also called WHOIS) limits queries to a single attribute and defines a resolution function that returns all objects that partially match the attribute. The NIC name server also enforces a restriction that an unambiguous attribute, called a *handle*, be registered for each object, such that if a client gives a handle, the naming system is guaranteed to return at most one object. Handles are implemented by attaching a unique prefix to a registered attribute so as to to ensure its uniqueness.

Enterprise generators are similar to the data mining techniques used by Netfind [15] and Fremont [4]. Data mining gathers information from several existing repositories and removes imperfections by correlating similar attributes. Enterprise generators are more general in that they provide a hook for gathering information and for determining its viability using techniques other than data mining. Enterprise views data mining as a technique that can be used for building useful mapping and agreement protocols.

7 Conclusion

This paper describes the Enterprise distributed user directory service. Enterprise is unique among user directory services for three reasons: *(i)* the system collects information automatically, i.e., user's are not required to maintain the information in the system; *(ii)* the system provides efficient searching for a large class of names without centralizing information, this respects the autonomy of participating organizations; and *(iii)* the system allows queries to be formed from arbitrary sets of attributes, there is no ordering of attributes and no special names that must be remembered.

Bibliography

[1] David Michael Betz. *Xscheme: An Object-oriented Scheme*. Peterborough, NH, 1989.

[2] Mic Bowman, Saumya Debray, and Larry L. Peterson. Reasoning about naming systems. *ACM Transactions on Programming Languages and Systems*, 1990. In Second Revision.

[3] Mic Bowman, Larry Peterson, and Andrey Yeatts. Univers: An attribute-based name server. *Software – Practice and Experience*, 20(4):403-424, April 1990.

[4] S. S. Coleman, D. C. M. Wood, and M. F. Schwartz. Fremont: A system for discovering network characteristics and problems. In *Proceedings of the Winter 1993 Usenix Conference*, San Diego, California, January 1993.

[5] David H. Crocker. Standard for the format of arpa internet text messages. Internet Request for Comments 822, Network Information Center, SRI International, Menlo Park, CA, August 1982.

[6] P. B. Danzig, S.-H. Li, and K. Obraczka. Distributed indexing of autonomous internet services. *Computing Systems Journal*, 5(4), 1992. To Appear.

[7] Debra Deutsch. An introduction to the X.500 series network directory service. Technical report, BBN Laboratories Incorporated, June 1988.

[8] International Organization for Standardization. *Information processing systems: open systems interconnection – the directory – overview of concepts, models, and service*, 1988. Draft International Standard ISO 9594-1:1988(E).

[9] B. Kahle and A. Medlar. An information system for corporate users: Wide area information servers. *ConneXions – The Interoperability Report*, 5(11):2-9, November 1991.

[10] S. A. Naqvi and R. Krishnamurthy. Database updates in logic programming. In *Proceedings of the Seventh ACM Symposium on Principles of Database Systems*, pages 251-262, 1988.

[11] Joann J. Ordille and Barton P. Miller. Nomenclator descriptive query optimization for large X.500 environments. In *sigcomm91*, August 1991.

[12] Larry L. Peterson. A yellow-pages service for a local-area network. In *Proceedings of the SIGCOMM '87 Workshop: Frontiers in Computer Communications Technology*, pages 235-242, Stowe, VT, August 1987.

[13] Larry L. Peterson. The Profile naming service. *ACM Transactions on Computer Systems*, 6(4):341-364, November 1988.

[14] J. Pickens, E. Feinler, and J. Mathis. The NIC-name server – a datagram based information utility. In *Proceedings 4th Berkeley Workshop on Distributed Data Management and Computer Networks*, August 1979.

[15] Michael F. Schwartz and Panagiotis G. Tsirigotis. Experience with a semantically cognizant internet white pages directory tool. *Journal of Internetworking – Research and Experience*, 2(1):23-50, March 1991.

[16] M. Solomon, L. Landweber, and D. Neuhengen. The CSNETname server. *Computer Networks*, 6:161-172, 1982.

[17] Gerald J. Sussman and Guy L. Steele Jr. Scheme: an interpreter for extended Lambda calculus. Technical Report Memo 349, MIT Artificial Intelligence, December 1975.

[18] D. Zimmerman. The finger user information protocol. Internet Request for Comments 1288, Network Information Center, SRI International, Menlo Park, CA, December 1991.

Author Information

Mic Bowman graduated with a Ph.D. in Computer Science from the University of Arizona in 1990. He is currently an assistant professor in the Penn State Computer Science department. You can reach him via U.S. Mail at Computer Science Department, 315 Whitmore Lab, Penn State University, University Park, PA 16802-6103. Reach him electronically at bowman@cs.psu.edu.

Chanda Dharap is currently pursuing her Ph.D at the Pennsylvania State University. She received her Masters degree in Computer Science also from Penn State in 1991. Her research interests include distributed systems and internetworking – in particular, problems associated with resource location and management in an internet environment. The author can be reached by email at chanda@cs.psu.edu and by U.S. Mail at 333 Whitmore Lab, University Park, PA 16802.

Essence: A Resource Discovery System Based on Semantic File Indexing

Darren R. Hardy & Michael F. Schwartz – University of Colorado, Boulder

ABSTRACT

Discovering different types of file resources (such as documentation, programs, and images) in the vast amount of data contained within network file systems is useful for both users and system administrators. In this paper we discuss the *Essence* resource discovery system, which exploits file semantics to index both textual and binary files. By exploiting semantics, Essence extracts keywords that summarize a file, and generates a compact yet representative index. Essence understands nested file structures (such as uuencoded, compressed, "tar" files), and recursively unravels such files to generate summaries for them. These features allow Essence to be used in a number of useful settings, such as anonymous FTP archives. We present measurements of our prototype and compare them to related projects, such as the Wide Area Information Servers (WAIS) system and the MIT Semantic File System (SFS). We demonstrate that Essence can index more data types, generate smaller indexes, and in some cases index data faster than these systems. Our prototype generates WAIS-compatible indexes, allowing WAIS users to take advantage of the Essence indexing methods.

Introduction

In the past two years, a number of resource discovery tools have been introduced to help users locate and use the massive amount of information available in the Internet [Schwartz et al. 1992b]. As disks have become larger, cheaper, and more plentiful, resource discovery has also become a problem in general purpose file systems, such as the Sun Network File System (NFS). Yet, the current set of Internet discovery tools do not apply well to such an environment, for three reasons.

First, information in general file systems is typically very irregularly organized. Most Internet data is explicitly intended for sharing, and hence people often put some effort into organizing the information into a coherent whole (e.g., placing an entire file system into a meaningful hierarchical directory in an *anonymous* FTP site). In contrast, most general file system data are organized according to the individual whims of many people. Therefore, resource discovery systems that depend heavily on users to organize and browse through data (such as Prospero, [Neuman 1992], WorldWideWeb [Berners-Lee et al. 1992], or Gopher [McCahill 1992]) do not work well for general purpose file system data. Instead, automated search procedures are needed. This typically means generating some type of index of the available information [Salton & McGill 1983].

Second, general file systems contain a range of different types of data, from unstructured text to structured data. Systems that use a generic indexing procedure (such as archie [Emtage & Deutsch 1992] or WAIS [Kahle & Medlar 1991]) produce larger or less useful indexes under these circumstances. For example, WAIS is most effective when used on ASCII text files. Using WAIS to index executables and other files found in general file systems is not very effective. The indexes tend to do a poor job of locating information, and tend to be quite large.

Third, Internet discovery tools typically focus on information known to be of reasonably broad interest. For example, anonymous FTP archives typically contain popular documents and software packages, which exhibit heavy sharing [Schwartz et al. 1992a]. In contrast, general-purpose file systems typically contain mostly user-specific data that exhibit relatively little sharing [Muntz & Honeyman 1992, Ousterhout et al. 1985]. Current Internet resource discovery tools have difficulties with such low sharing-value data. For example, WAIS's full-text indexing mechanism may locate many uninteresting files if applied to an entire general file system, since keywords will be generated from files that are of interest to few users.

In this paper we present a system for supporting resource discovery in general purpose file systems. The system addresses the above problems by generating indexes based on an understanding of the semantics of the files it indexes. This technique supports compact yet representative summaries for general collections of data. In addition to supporting file indexes, summaries can be browsed to help decide whether to retrieve a file across a slow network. We call our system *Essence* because of its ability to summarize large amounts of data with relatively small indexes.

We begin with a discussion of indexing techniques. We then survey previous work related to

semantic indexing. We discuss how Essence accomplishes semantic indexing and uses it as a basis for resource discovery. Finally, we discuss the details of our prototype, and present some measurements that compare Essence with WAIS and SFS.

Full Text vs. Filename vs. Semantic Indexing

WAIS supports fine-grained information access by building full-text indexes, in which every keyword from a textual document appears in the index. As indicated above, this approach is primarily useful for purely textual, widely popular data. Moreover, WAIS has large space requirements: its indexes are comparable in size to the data files they represent. Because of these space requirements, WAIS distributes the indexes among the hosts that provide data.

A less space intensive indexing approach is used by archie, in which anonymous FTP files are summarized by name only (i.e., archie indexes contain no information derived from file content). This approach produces indexes that are roughly one thousandth the size of the data that they represent. In turn, this compact representation allows a great deal of index information to be collected onto a single machine, supporting far-reaching searches (currently reaching over 1000 archive sites). Yet, because archie indexes contain only filenames, they support only name-based searches. Searches based on more conceptual descriptions of resources are not possible, except when the filenames happen to reflect some of these conceptual descriptions.

The range of structure and the low overall sharing value in general purpose file systems (as discussed in the introduction), coupled with the need for conceptual descriptions and the need for compact indexes (motivated above), all suggest the use of a different means of indexing data. That means is *semantic* indexing.

Semantic indexing involves analyzing the structure of file data in different ways, depending on file type. For example, UNIX manual page files are broken into structured sections from which it is possible to extract information about a program's name and description, a usage synopsis, related programs or files, and author information. By generating information for different types of files in different manners, semantic indexing can generate representative keywords without including every word from a file. In addition to saving space, this technique can avoid including keywords that might muddle the quality of an index. For example, it makes little sense to include C language constructs like "struct" when indexing C source code, since these keywords do not distinguish the conceptual content of different C programs.

Semantic indexing involves two stages. The *classification* stage identifies promising files to index within a file system,[1] as well as type information for each identified file. The *summarizing* stage applies an appropriate indexing procedure (called a *summarizer*, to emphasize the space reduction characteristic) to each identified file, based on the type information uncovered in the classification stage.

Since summarizers understand file types, they can extract keyword information for both textual and binary files. For example, many binary executables have related textual documents describing their usage, from which keyword information can be extracted.

Since keyword information is extracted based on knowledge of where high-quality information might be located, semantic indexing extracts fewer keywords than full-text indexing, and thus generates smaller indexes. Yet, it retains the fine-grained, associative access capability of full-text indexes.

The Essence System

Essence provides an integrated system for classifying files, defining summarizer mechanisms, applying appropriate summarizers to each file, and traversing a portion of a file system to produce an index of its contents.

Essence determines file types by exploiting file naming conventions (such as common filename extensions like ".c"), and locating identifying data or common structures within files (such as UNIX "magic numbers"). Once Essence determines a file's type, it executes the appropriate summarizer to extract keywords from the file. Among other types, Essence understands *nested* file structures, such as compressed, uuencoded "tar" files. It recursively extracts files *hidden* within a nested file, and indexes them.

As a design goal, we wanted to allow summarizers to be constructed quickly and easily, so that Essence could be made to understand many different file types, and so individual sites could customize their summarizers. To accomplish this goal, we allow summarizers to be as simple as a one line script (perhaps containing a "grep" command).

Essence indexes can allow users to locate needed data. Moreover, Essence produces *summaries* of file data, which allow quick perusal of potentially large files.

Essence has many practical resource discovery applications:

- Systems administrators and users can use Essence to locate and learn about resources

[1] For example, this procedure might embody site-specific knowledge that certain parts of the file tree contain uninteresting administrative information, and hence needn't be indexed. Our current prototype does not select file system subsets - it simply indexes whatever file trees are specified.

contained within their file systems without understanding the details of their local environment. This is particularly helpful in environments where mount points are "hidden" by the *amd* auto-mount system.
- Public archive administrators can use Essence to index archive contents, providing compact yet representative descriptions of files, including compressed archives. These indexes allow users to search for information more effectively, and examine summaries about files before retrieving them.
- People who wish to index data and search it using WAIS can use Essence to index more file types than WAIS itself currently supports, and to produce more space efficient indexes.

Once Essence generates an index for a portion of a file system, it exports its indexes via WAIS's search and retrieval interface. This allows our indexes to be used within the context of a well established, easy to use information system.

Related Work

Identifying and Locating File Resources

Semantic indexing depends on successfully determining file types. Furthermore, Essence uses semantic indexing to locate file resources. Many systems can either determine file types or locate file resources, but Essence integrates both aspects into a single system.

- The Modules system is a sophisticated administrative approach to locating file resources associated with specific applications [Furlani 1991]. Applications are associated with a particular *module*, which can be easily incorporated or removed from a user's environment. Both the location and identification of the applications and their file resources are explicitly supplied by an administrator, and are hidden from the user.
- The NeXT file system browser determines common file types by exploiting filename extensions [NeXT 1991]. It then displays an icon representative of the file's type. Users can *launch* a specific application by supplying only a filename, as the application that is launched is determined by the file's type. Locating file resources is accomplished by browsing a UNIX file system hierarchy.
- The UNIX *file* command attempts to determine various file types based on file contents, but provides no mechanism for locating files [USENIX 1986].
- The UNIX *find* command locates files using an exhaustive search of a portion of a file system. It allows predicates to be specified concerning which files to locate. Among other things, these predicates can specify location based on the file types understood by the UNIX file system (such as ordinary file, directory, or symbolic link) [Leffler, et al. 1989, USENIX 1986]. Higher-level types (such as image, script, or C source code) cannot be specified.
- Many programs use file naming conventions to infer file types. C compilers, for example, assume that a filename ending in ".c" is a C source file, while a file ending in ".o" is a relocatable object file. Similarly, *make* has various implicit rules based on file naming conventions.

Exploiting File Semantics

Semantic indexing also depends on the ability to extract good keyword information from files based on their file types. A number of UNIX commands can extract information with varying degrees of quality from files based on their file types [USENIX 1986].

- *ctags* extracts procedure, macro, and variable names from C source and header files. Some versions of *ctags* understand other programming languages, such as Lisp, Pascal, and C++.
- *strings* extracts embedded ASCII text strings from binary files.
- *deroff*, *detex*, and *ps2txt* extract ASCII text from troff, TeX, and PostScript files, respectively.
- *what* extracts embedded Source Code Control System (SCCS) information from files.

Essence provides a single cohesive system that integrates determining file types, locating file resources, and exploiting file semantics to extract good keyword information from files.

MIT Semantic File System

The MIT Semantic File System (SFS) uses semantic file indexing to provide a more effective storage abstraction than traditional hierarchal file systems [Gifford et al. 1991]. SFS exploits filename extensions to determine file types, and then runs *transducers* on files to extract keyword information for building an index. SFS provides a *virtual directory* interface to search the resulting index and to access files. Virtual directory names are interpreted as queries against the index, and the contents of virtual directories are the results from these queries. Therefore, users perceive a search-based interface to explore file systems, rather than the more traditional hierarchical file system interface.

Although Essence and SFS use similar semantic indexing techniques, they differ in orientation, summarizer breadth, and space efficiency.

Orientation

SFS emphasizes semantic indexing as a storage abstraction. In contrast, Essence emphasizes semantic indexing as a basis for resource discovery. Concretely, while both systems support flexible associative access to file data, they export the data

differently. Essence exports the data through a search and retrieval interface, while SFS exports the data through a file system interface. The advantage of the SFS approach is that it reuses an existing and familiar storage abstraction. The disadvantage is that doing so leads to undefined semantics. For example, if a user tries to copy data into a virtual directory (created as a result of an SFS query), the semantics are undefined.

Summarizer Breadth

Essence summarizers are autonomous UNIX programs, which are easy to implement, integrate, and maintain. The Essence prototype implements summarizers for many more file types than SFS does. Essence can index a wide variety of textual and binary data common in network file systems.

Space Efficiency

The Essence prototype provides better index compression than the SFS prototype. Comparative measurements appear later in this paper.

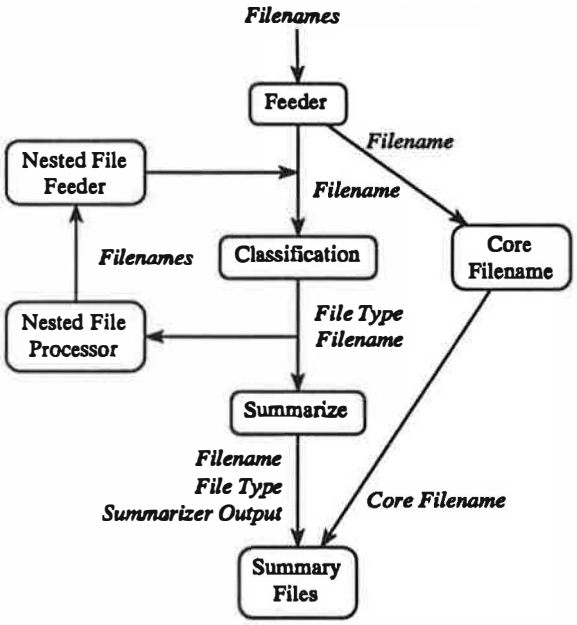


Figure 1: Organization of the Essence System

The Design of Essence

Figure 1 shows how Essence is organized. Essence operates as follows:

- Users supply Essence with the filenames from a select portion of a file system that they wish to index.
- The *Feeder* module iteratively passes each of these filenames to the *Classification* module, which determines the file's type.
- The *Summarize* module chooses an appropriate summarizer based on the file's type. It then runs this summarizer on the file to extract keywords for the *Summary Files*.

The three modules, *Core Filename*, *Nested File Processor*, and *Nested File Feeder*, allow Essence to support nested files.

- Essence saves the initially supplied filename as the *Core Filename*.
- If the *Classification* modules determines that the file has a nested file type (such as a compressed file), it passes the file to the *Nested File Processor*.
- The *Nested File Processor* extracts the hidden files from the nested file structure and passes the extracted files to the *Nested File Feeder*.
- The *Nested File Feeder* module performs the identical function as the *Feeder* but bypasses the *Core Filename* module.

Determining File Types

Essence determines file types using a combination of exploiting file naming conventions and heuristically locating identified data and common structures within files.

Exploiting File Naming Conventions

Observing even simple conventions in file naming can determine file types with fairly high certainty. The most basic file naming convention is filename extensions. For example, filenames with a ".c" extension are typically C source code files; filenames with a ".ps" extension are typically PostScript image files; and filenames with a ".txt" extension are typically ASCII text files. File naming conventions also include using specific words within a filename. For example, information about an entire source distribution or application is often found in files whose name contains the string "README". Files named "Makefile" are typically associated with the UNIX *make* command [USENIX 1986].

In Essence, file naming conventions are represented as regular expressions. For example, **.ps* or **[Mm]akefile** represent the PostScript and Makefile file types, respectively. Expressing file naming conventions as regular expressions allows sites to easily integrate their local semantics into Essence.

Locating Identifying Data and Common Structures

In addition to using naming conventions, Essence examines file contents to try to determine file types. In particular, many files have an identifying *magic* number associated with them. For example, NeXT binary executables start with the hexadecimal number *0xfeedface*, and Sun Pixrect images start with the hexadecimal number *0x59a66a95*. Furthermore, common structures within a file may determine its file type. For example, PostScript images start with the string "%!"; UNIX shell programs start with the string "#!"; C source code files typically have comments denoted with the "/* */" delimiters; electronic mail files have distinctive header tags, such as *From:*, *Received by:*, and

Sender:; and USENET news articles also have distinctive header tags, such as *Newsgroups:*, *Distribution:*, and *Path:*.

As with exploiting file naming conventions, locating identifying data and common structures within a file is a rule-based technique expressed with regular expressions. Sites can easily integrate their local semantics into the discovery process by modifying these rules.

Nested File Structure

Nested files contain *hidden* files. Examples include compressed files, tar files, uuencoded files, ZIP files, and shell archive files. Furthermore, files can be arbitrarily nested within these file types. For example, compressed tar files or uuencoded compressed files are common. Understanding nested file structures is useful in file system environments (such as anonymous FTP file systems) in which the vast majority of files have nested structure.

When Essence determines that a file has nested structure, it extracts the hidden files, determines the resulting files' types, and summarizes them. This process continues recursively, until no more nesting is found. Extracting hidden files from a nested file is accomplished by running a corresponding extraction program, such as the UNIX *uncompress* command for compressed files, the UNIX *tar* command with the 'x' flag for tar files, or the UNIX *uudecode* command for uuencoded files.

Summarizers

Essence's summarizers are simple stand-alone UNIX programs that are easy to write and integrate into the system. This design provides a powerful paradigm for exploiting file semantics. Each summarizer is associated with a specific file type, and understands the file's format well enough to extract summary information from the file. For example, the summarizer for a UNIX troff-based manual page understands the troff syntax and the conventions used to describe UNIX programs. It uses this understanding to extract summary information, such as the title of a program, related programs and files, the author(s) of the program, and a brief description of the program. Similar techniques can be used on many other moderately structured file types, such as source code. However, some file types do not easily lend themselves to automated interpretation. For example, plain ASCII text files typically contain unstructured data that is difficult to exploit effectively. Similarly, the UNIX *ps2txt* program can extract ASCII text from PostScript images, but the resulting information is unstructured text.

Essence Prototype

In this section, we describe the techniques used by the Essence prototype to determine file types and exploit file semantics with summarizers. We also discuss how we integrated Essence with WAIS.

Determining File Types

As described earlier, file types are determined by understanding naming conventions and locating identifying data and common structures within a file. In the prototype, naming conventions are expressed with case-insensitive regular expressions. The following example shows some entries from the configuration file that holds the expressions. In this file, the first field is the file type, and the second field is a case-insensitive regular expression for the corresponding file naming convention.

```
Compressed      .*\.Z
ManPage         .*\.[12345678]
PostScript      .*\.(ps|eps)
README          .*readme.*
SCCS            s\..*
```

The prototype also uses the UNIX *file* command to determine file types, based on identifying data and common structures within a file [USENIX 1986]. *file* uses the */etc/magic* file to specify recognizable file types. The following list shows some sample entries from */etc/magic*, where the first field is the offset of the identifying data or common structure, the second field is the type this data, the third field is the identifying data or common structure itself, and the last field is the corresponding file type.

```
0   string  /*              C program text
0   string  \037\235        Compressed data
0   long    0xfeedface      NeXT binary pgm
0   string  #!/bin/perl     Perl program
0   string  %!              PostScript image
```

Creating a suitable *magic* file is not trivial, because the identifying data or common structures must be distinctive. For example, the "/*" delimiter for C programming language comments is not sufficiently distinctive, and will likely appear in a variety of types of files. A lack of distinctive identifying data or common structures is common for binary formats, which usually depend on a single magic number. Although distinctive *magic* entries are difficult to formulate, careful selection of a *magic* file allows *file* to accurately identify file types. In Essence, building the *magic* file was accomplished through experimentation with various entries.

Summarizers

In the prototype, summarizers are simple UNIX programs that extract keyword information through understanding the syntax and semantics of a specific file type. Currently, the prototype supports summarizers for twenty-one file types and four nested file types. Table 1 describes these file types, their frequencies of occurrence by number of files, average file size, and which systems support them in two file system environments: an NFS file system that contains commonly shared data and programs in our local environment, and a fairly popular anonymous FTP file system (ftp.cs.colorado.edu). The most frequent file types in the NFS file system were *Text*,

CHeader, and *ManPage*. In the anonymous FTP file system the most frequent file types were *CHeader*, *C*, and *Text*.

Essence supports more of the file types found in common NFS and anonymous FTP file systems than either WAIS or SFS, as shown in Table 1. Although WAIS and SFS support most of the frequently occurring file types (such as *Text*, *C*, and *CHeader*), Essence is the only system that supports the file types that contribute most to overall data size (such as *Binary*, *Tar*, and *Archive*). Occurrence frequencies will be used in our measurements, later in this paper. Note that Table 1 does not list specialized file types supported by WAIS or SFS that are not supported by Essence, because those types do not occur in common NFS and anonymous FTP file systems (and hence we have no measurements for them). Examples include MedLine and New York Times formats. There are 12 such formats understood by WAIS, and 4 understood by SFS. Also, as indicated in the table, SFS indexes Unknown file types. It does so by including the standard UNIX attributes in the index, such as owner, directory, and group.

Table 2 briefly describes the techniques used by the Essence summarizers for the supported file types, other than nested types (the techniques for which were already discussed, in the "Nested File Structure" section). Many other potential summarizers are possible. For example, writing summarizers for other types of source code (such as Lisp or Pascal) would be an easy extension of the prototype's source code summarizers. However, writing summarizers for audio or image formats would be difficult.[2]

The following sections describe some of the techniques used in various summarizers, representative of Essence's supported file types.

[2]One possibility would be to sample a bitmap file down to an icon. While this does not easily support indexing, it could be used to support quick browsing before retrieving an entire image across a slow network.

File Type	File Type Description	File Types Supported By			Frequency by Number of Files		Average File Size	
		Essence	WAIS	SFS	NFS	AFTP	NFS	AFTP
Archive	Library archives	×		×	0.36	0.12	626.31	47.52
Binary	Binary Executable	×		×	5.06	0.02	145.74	112.00
C	C source code	×	×	×	1.27	19.33	3.87	28.36
CHeader	C header files	×	×	×	14.73	22.42	4.29	2.40
Command	UNIX shell scripts	×	×		1.78	3.06	2.75	1.55
Compressed	Compressed file	×			0.69	11.81	114.98	73.29
Directory	Directory	×		×	4.87	5.05	0.81	0.50
Dvi	Device-indep. TeX output	×	×		0.03	0.87	33.32	59.18
Mail	Electronic mail	×	×	×	0.02	0.17	1.79	35.30
Makefile	UNIX makefiles	×	×	×	0.26	3.87	0.85	3.04
ManPage	UNIX manual pages	×	×	×	13.78	0.70	6.76	295.92
News	USENET news articles	×	×	×	0.01	0.04	21.96	1.75
Object	Relocatable object file	×		×	0.00	1.12	0.00	28.11
Patch	File difference listing	×	×		0.02	0.00	1.88	0.00
Perl	Perl script	×	×		0.00	0.02	0.00	3.62
PostScript	PostScript images	×	×		1.42	3.31	64.64	194.45
RCS	RCS version control files	×	×		0.00	1.41	0.00	8.41
README	High-quality information	×	×	×	0.38	1.32	1.95	2.88
SCCS	SCCS version control files	×	×		0.00	0.00	0.00	0.00
ShellArchive	Bourne shell archive	×			0.00	0.10	0.00	486.75
Tar	Tar archive	×			0.00	0.81	0.00	1734.21
Tex	TeX source docs	×	×	×	0.67	0.23	17.79	34.17
Text	Unstructured ASCII text	×	×	×	21.64	19.73	7.87	31.11
Troff	Troff source docs	×	×		0.03	0.25	9.21	9.08
Unknown	Unknown file type			×	32.96	4.26	44.02	16.10

Table 1: Supported Common File Types and their Frequency and Average Size in Measured File Systems

Directory

Obtaining a listing of the files in a directory is an obvious method for a *directory summarizer*. However, Essence strives to obtain a higher-level understanding of a directory's contents. Therefore, the prototype attempts to extract copyright information from files, in addition to the directory listing. Copyright information typically contains project, application, or author names. Keyword information from README files is also included in the directory summarizer, since these files contain high-quality information about the directory's contents.

File Type	Summarizer Description
Archive	Extract symbol table
Binary	Extract meaningful strings, and manual page summary
C	Extract procedure names, #include'd filenames, and comments
CHeader	Extract procedure names, included filenames, and comments
Command	Extract comments
Directory	Extract directory listings, copyright information, and README files
Dvi	Convert to ASCII text
Mail	Extract select header fields
Makefile	Extract comments
ManPage	Extract author, title, etc., based on "-man" macros
News	Extract select header fields
Object	Extract symbol table
Patch	Extract filenames
Perl	Extract procedure names and comments
PostScript	Convert to ASCII text
RCS	Extract RCS-supplied summary
README	Use entire file
SCCS	Extract SCCS-supplied summary
Tex	Convert to ASCII text
Text	Extract first 100 lines
Troff	Extract author, title, etc., based on "-man", "-ms", "-me" macro packages, or extract section headers and topic sentences.

Table 2: Essence Summarizer Techniques

Binary

An obvious method for a *binary summarizer* is to extract ASCII strings from the binary file, using the UNIX *strings* command. However, Essence filters these extracted ASCII strings using heuristics that only keep strings that convey the binary's purpose, such as usage, version, or copyright information. Essence also uses cross references to obtain high-quality summary information from binary executables. For example, the binary summarizer looks for associated manual pages for the given binary executable, and generates keywords using the *manual page summarizer* on it.

Formatted Text

Although formatted text (such as TeX, Troff, or Word Perfect) has structured syntax, effectively summarizing these files is difficult unless semantic information is also available [Knuth 1984, Lamport 1986, USENIX 1986]. For example, plain Troff files or Troff files using the "-me" macros are difficult to exploit semantically, since their syntax is associated with formatting commands (such as font size or line spacing), rather than more conceptual commands (such commands to indicate an author's name or paper title). Troff files using the "-ms" or "-man" macros are much easier to summarize, since they contain conceptual commands (such as delimiting an abstract, author, and title).

Essence supports a sophisticated summarizer for Troff and the "-me", "-ms", and "-man" Troff macros. The *TeX summarizer* only extracts ASCII text from TeX files using *detex*, but exploiting TeX semantics would be a trivial extension of the methods used in our *Troff summarizer*.

Simple Text

Simple text is difficult to summarize because it is unstructured. Essence assumes that the highest quality information in most unstructured text files is near the beginning of the file, as is common with paper abstracts or tables of contents. Therefore, the *text file summarizer* extracts keywords from the first one hundred lines of the text file. However, README files typically contain crucial, concise information about a distribution or application. Using a full-text index of README files provides high-quality keywords without occupying too much space. Therefore, the *README summarizer* uses the entire file to generate keywords.

The *Dvi*, *PostScript*, and *Tex summarizers* extract keywords from all of the ASCII text extracted from these files. Essence assumes that these file types contain generally useful information, and hence generates full text indexes for them.

Source and Object Code

Both source and object code are highly structured, and contain easily exploited semantic information. The *C summarizer* extracts procedure names, header filenames, and comments from a C source code file. Similarly, the *object summarizer* extracts the symbol table from an object file.

WAIS Interface

Essence exports its indexes through WAIS's search and retrieval interface, allowing users to use tools such as *waissearch* and the X Windows-based graphical user interface *xwais*. In order to generate

WAIS-compatible indexes, Essence uses WAIS's indexing software to index the Essence summary files. This mechanism generates full-text WAIS indexes from the Essence summary files.

We modified the WAIS indexing mechanism to understand the format of the Essence summary files, so that it generates meaningful WAIS *headlines*. These headlines provide users with a short description of a single file, usually a filename. With Essence, headlines represent a file's core filename, its actual filename, and its file type.

To support additional file types, WAIS must be recompiled with new procedures that understand these file types. With Essence, one need only write a new summarizer, add its name to a configuration file, and add new heuristics for identifying the file type; no recompilation is necessary. In this sense, Essence modularizes the typed-file indexing extensions that WAIS can use, because it removes the keyword extraction process from WAIS and places it instead in Essence. Essence is better suited to incorporating new file types, and can be quickly adapted to become a *comprehensive* indexing system.

Figure 2 shows an example search of an index generated by Essence of the ftp.cs.colorado.edu anonymous FTP file system. It shows an ordered list of the ten files that best match the keyword *netfind*.[3] The headlines have up to three fields representing the matching file: the core filename, the filename (if different from the core filename), and the file type.

[3] *Netfind* is an Internet user directory service [Schwartz & Tsirigotis 1991].

Consider the effectiveness of the example search in Figure 2. The best match is a PostScript paper that discusses a number of techniques for distributed information systems, with particular emphasis on techniques demonstrated by Netfind; the second match is the same file, but found in the compressed tar distribution *ALL.PS.tar.Z*. The third match is the C source code for the interactive user interface to Netfind. The fourth match is the README file found in the Netfind distribution directory; the fifth match is the same file, but found in the compressed tar distribution *netfind.3.10.tar.Z*. The sixth match is the UNIX manual page for Netfind. The remaining matches are PostScript papers in which Netfind is discussed.

In WAIS, a user retrieves files by selecting a matching headline. With Essence, if the headline represents a file hidden within a nested file (such as the first headline in Figure 2), the summary file is retrieved, instead of retrieving the hidden file itself. If the headline represents a plain file (such as the fourth headline in Figure 2), the file is retrieved. This functionality requires allocating storage for both the required summary files and the index. However, it allows users to browse through remote file systems by retrieving and viewing small summary files without having to retrieve complete files. This is useful when trying to decide whether to transfer large files across a slow network.

Evaluation and Measurements

In this section, we present measurements of indexing speed and space efficiency, for Essence, WAIS, and SFS. We also discuss the usefulness and

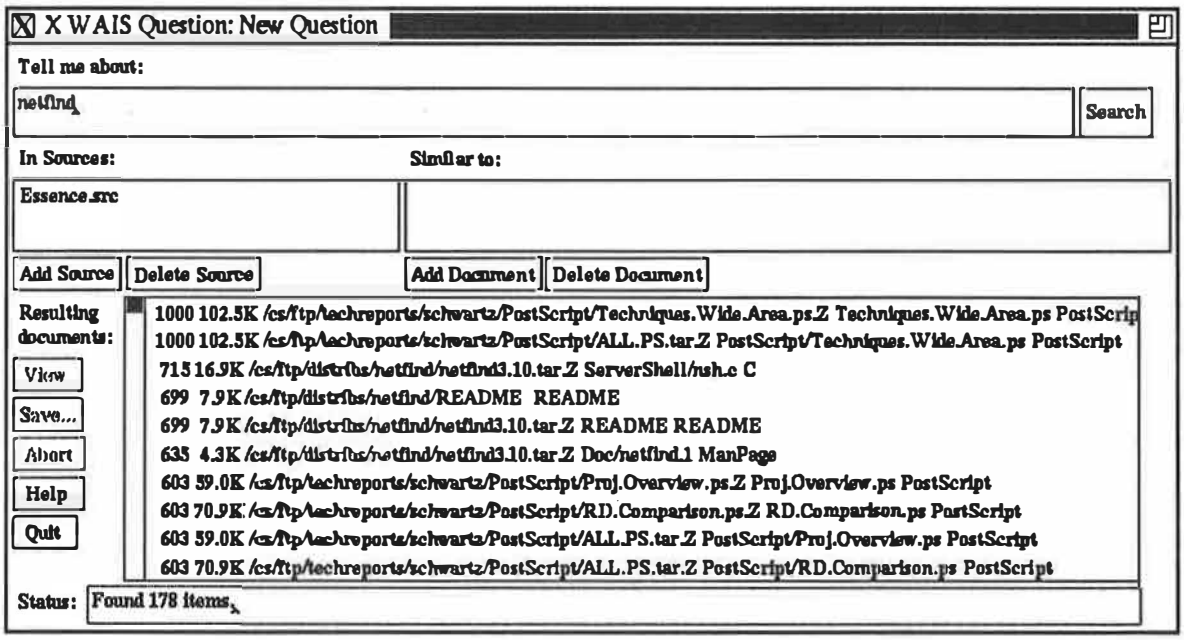


Figure 2: Example WAIS Search Using Essence-based Index

overhead of indexing nested files. Finally, we discuss the difficulties in evaluating keyword quality.

Before presenting measurements of the various systems, we note that it is difficult to interpret time and space efficiency measurements of the systems being compared, for two reasons. First, indexing speed and compression are highly dependent on indexing techniques. For example, an indexer that skips most of the data (such as our Text summarizer) will achieve much higher indexing speeds and compression factors than one that uses all of these data (such as the Text indexers used by SFS and WAIS). In this case, the salient issue becomes recall/precision effectiveness of the generated indices (which is difficult to quantify). For example, a small, quickly generated index would not be a reasonable tradeoff if one could not use this index to locate desired data. Second, aggregate measurements (as given in Table 4) are affected by the distribution of different file types in the sample file systems. Ideally, we would have measured each indexing system against the same file system data. We did this for WAIS and Essence, but the SFS code was not available at the time we made these measurements. Instead, we attempted to interpret the measurements given in [Gifford et al. 1991]. Notwithstanding these difficulties in interpreting the measurements, we feel it is worthwhile to present quantitative comparisons of these systems.

Table 3 presents the space and time measurements for Essence and WAIS, based on file type. We do not show measurements for nested file types

File Type	Indexing Rate (KB/min)		Compression Factor vs. Index		Semantic Exploitation
	Essence	WAIS	Essence	WAIS	Overhead
Archive	3289.18	-	10.89	-	*low*
Binary	563.40	-	21.15	-	*high*
C	357.84	593.15	2.46	1.45	*high*
CHeader	164.87	342.93	1.33	0.65	*high*
Command	168.20	277.30	1.23	0.43	*high*
Dvi	278.71	2160.00	0.79	1.76	*high*
Mail	3718.12	1071.89	9.44	0.74	*low*
Makefile	421.05	648.65	2.33	0.86	*high*
ManPage	165.67	661.59	3.34	1.18	*high*
News	1913.29	329.24	20.82	0.63	*low*
Object	2588.75	-	15.93	-	*low*
Patch	7218.00	993.30	80.20	2.00	*low*
Perl	282.50	713.68	2.05	0.88	*high*
PostScript	1151.19	765.60	4.56	1.67	*low*
RCS	1293.16	614.25	5.91	1.27	*low*
README	390.00	400.83	0.68	0.65	*high*
SCCS	315.79	1500.00	3.12	3.52	*high*
Tex	598.93	385.52	2.09	0.92	*low*
Text	4699.59	1346.67	8.13	1.37	*low*
Troff	703.01	2036.92	7.04	1.97	*high*

Table 3: Essence and WAIS Time and Space Measurements Based on File Type.

File System	Indexing Rate (KB/min)			Compression Factor vs. Index		
	Essence	WAIS	SFS	Essence	WAIS	SFS
NFS	1489.58	891.40	712.00	14.40	1.27	6.82
Anonymous FTP	1826.17	897.01	712.00	4.93	1.44	6.82
Average	1657.88	894.21	712.00	9.67	1.35	6.82

Table 4: Weighted Time and Space Averages Based on File Type Frequencies

here. Those measurements are discussed in Table 6. Nor do we show measurements for SFS in this table, because transducer-specific information was not available. Also, note that the indexing costs shown for Essence include the time and space needed to indices - not just the summaries that are produced as an intermediate step. As indicated in Table 1 and with a '-' in Table 3, WAIS and SFS cannot index all of the file types that Essence can. Table 3 shows that because there is a high amount of overhead associated with interpreting the semantics of a file type, Essence indexes slower than WAIS for some file types. Essence indexing is faster than WAIS for file types that have a low amount of semantic interpretation overhead.

Table 4 presents weighted averages of the space and time measurements in Table 3, based on the file type frequencies and average file sizes (as measured in Table 1). The weighted averages were computed using the formula:

$$\sum_{i=0}^{n}(f_i a_i)v_i,$$

where f_i is the frequency associated with file type i, a_i is the average file size associated with file type i, v_i is the the indexing rate or the compression factor value from Table 3 associated with file type i, and n is the number of file types supported by the system. $f_i a_i$ is used to normalize the measurements, to reflect only the system's supported file types. In particular, only non-nested file types are included in the aggregate measurements for WAIS and SFS (since those systems do not support nested files), while all types (including nested files) are included in the Essence measurements. We discuss the "unraveling" costs of dealing with nested file structures in Table 6.

The Essence and WAIS measurements were performed on a Sun Microsystems 4/280 server running SunOS 4.1.1, with a local SMD disk. The SFS measurements were performed on a Microvax-3 running UNIX version 4.3BSD [Gifford et al. 1991]. This machine is approximately one-third as fast as the Sun 4/280.

Table 4 shows that Essence can index data faster than WAIS. Taking into account the slower machine on which SFS was measured, SFS appears to index data somewhat faster than Essence does.

Essence obtains about a 10:1 index compression factor on the file types that it supports, compared to WAIS (1:1), SFS (7:1), and archie (765:1) [Emtage & Deutsch 1992]. These measurements are not perfect, because detailed SFS measurements were not available.

Table 5 shows the percentage of data in the measured file systems that Essence, WAIS, and SFS were successfully able to interpret and index. The NFS file system contained many custom file formats that the indexing systems were unable to interpret. However, the anonymous FTP file system contained many more common file formats. Even though Essence only supports a relatively small number of common file types (21), it can index 75% of the data found in an average file system – far greater than WAIS (33%) or SFS (18%).

We found that seventy-eight percent of the files in our anonymous FTP had nested structure. These measurements indicate that supporting nested file structures is essential for such file systems. In contrast, only one percent of the files in the NFS file system had nested structure. In the future nested file structures may become less common, as they mostly represent inadequacies of current file systems and remote access protocols. For example, tar files are popular in FTP file systems because they make it easier to retrieve an entire directory tree, and FTP does not provide a recursive retrieval mechanism.

	Essence	WAIS	SFS
Anonymous FTP	98.51	48.47	27.56
NFS	50.70	17.88	8.11
Average	74.61	33.18	17.84

Table 5: Percentage of Interpretable Data

Table 6 shows how much overhead the prototype incurs when indexing nested files in the measured anonymous FTP file system. In this table, the *Original Data* row concerns the data which reside in the anonymous FTP file system. The *Processed Data* row concerns the data that Essence processes while indexing the *Original Data*. These data include all of the original files and each file within a nested file structure. For example, given the file *foo.tar.Z* from the example in the previous Nested File Structure section, *foo.tar.Z*, *foo.tar*, *foo.c*, *foo.h*, *Makefile*, and *README* are all included in the *Processed Data*. The *Summarized Data* row concerns the data on which summarizers are run. For example, *foo.c*, *foo.h*, *Makefile*, and *README* are all included in the *Summarized Data*. The *Summary Output* row concerns the resulting summary files. The resulting index of the summary files consumed 12.94 megabytes.

Note that this compression ratio (60.22/12.94) understates the actual compression, because the indexed data actually consumed 262.03 MB. In particular, indexing systems (like WAIS) that do not support nested structure would have to leave the data uncompressed. Hence, we actually achieve a two-fold space reduction. WAIS would need to keep the uncompressed data around, and then would generate an index whose size was comparable to the uncompressed data. Essence generates a smaller index, and can function with compressed data. Putting the numbers together, WAIS would require approximately 264 MB of space for the uncompressed data plus index (basically, twice the size of the Summarized data), while Essence

requries only 73 MB total – a 72% space savings over WAIS.

Analysis of Keyword Quality

Qualitative analysis of information retrieval systems is difficult. *Recall/precision* measurements are difficult to obtain, since they rely on hand-chosen reference sets [Salton & McGill 1983], and hence do not scale well to measuring large information collections. More effective measurements might be obtained by evaluating the effectiveness of a system from experience with an active user community. We have made the Essence prototype is publically available to allow users to make their own subjective judgements.

	Total Number of Files	Total Size (in MB)
Original Data	1213	60.22
Processed Data	6409	262.03
Summarized Data	5334	132.36
Summary Output	5334	15.87

Table 6: Nested File Structure Overhead

Future Directions

On-the-Fly Nested File Summarizers

The Essence prototype relies heavily on the file system to implement nested file structure interpretation. This implementation degrades performance when indexing files with nested file structures (as shown in Table 6), because it causes a large amount of disk I/O. An in-memory implementation would significantly improve performance, by drastically cutting file system I/O. We are currently considering such an implementation, based on the GNU "tar" program, which supports an option to output extracted files to stdout.

Summarizers

The prototype currently supports twenty-one summarizers. Expanding Essence's summarizer suite to support more file types would further increase its effectiveness.

Anonymous FTP Indexing

Currently, the Essence index for the anonymous FTP site at ftp.cs.colorado.edu is available through WAIS using the *aftp-cs-colorado-edu.src* WAIS source. However, we would like to make more anonymous FTP sites available through WAIS, using Essence-based indexes. Using Essence to index public archives allows remote users to search information based on conceptual descriptions and to view summaries before retrieving files. This would help decrease the network traffic of unwanted files.

Record-Level Indexing Support

WAIS supports indexing and retrieving information with record-level granularity (e.g., allowing a file containing many electronic mail messages to be treated as a sequence of mail records). Essence only supports indexing and retrieving information with file-level granularity. A future improvement would be to modify Essence to support record structured files.

File Tree Pruning

The design of Essence's file classification stage includes the ability to identify promising files to index within a file system, in addition to type information for each file. Our current prototype does not select file system subsets – it simply indexes whatever file trees are specified. A future improvement would be to add selection criteria to the prototype (e.g., pruning files from consideration based on their location in the name tree, names/types, or sharing history). This would further refine the quality of indexes, and reduce the space required for indexing an entire file system.

Summary

The increasing abundance of inexpensive local disks creates resource discovery problems even in locally distributed file systems. The Internet resource discovery tools that have achieved popular acceptance over the past two years are not well suited to general purpose file systems, because of the irregular organization, the range of different degrees of information structure, and the generally low sharing value of information in such file systems.

In this paper we presented the Essence system, which generates file summaries based on an understanding of the semantics of the various types of files it indexes. The summaries are useful both for producing searchable indexes, and for allowing users to retrieve and browse small summaries before deciding whether to retrieve a large file across a slow network. Simple techniques to exploit file semantics yield compact yet representative indexes for both textual and binary files. The indexes generated in this fashion are more content-rich than archie's index, yet more space efficient than WAIS indexes.

Essence provides an integrated system for classifying files, defining summarizer mechanisms, applying appropriate summarizers to each file, and traversing a portion of a file system to produce an index of its contents. Importantly, Essence understands nested file structures (such as uuencoded, compressed, "tar" files), and recursively unravels such files to generate summaries for them. The ability to index nested files and many other file types allows Essence to be used in a number of useful settings, such as anonymous FTP archives.

Essence can index more data types, index data faster, and generate smaller indexes than WAIS or the MIT Semantic File System. Our prototype generates WAIS-compatible indexes, allowing WAIS users to take advantage of the Essence indexing methods.

Prototype Availability

The Essence prototype, including its source code and WAIS modifications, is publically available by anonymous FTP from ftp.cs.colorado.edu in /pub/cs/distribs/essence. The prototype is written in the C and Perl programming languages [Kernighan & Ritchie 1988, Wall & Schwartz 1991].

Acknowledgements

This material is based upon work supported in part by the National Science Foundation under grant NCR-9105372, and a grant from Sun Microsystems' Collaborative Research Program.

We thank Sean Coleman, Jim O'Toole, David Wood, and the USENIX program committee for their helpful comments on this paper.

References

[Berners-Lee et al. 1992] T. Berners-Lee, R. Cailliau, J. Groff and B. Pollermann, World-Wide Web: The Information Universe, *Electronic Networking: Research, Applications and Policy,* 1(2), Meckler Publications, Westport, CT, Spring 1992.

[Budd & Levin 1982] T. A. Budd and G. M. Levin, A UNIX Bibliographic Database Facility, Tech. Rep. 82-1, Department of Computer Science, University of Arizona, Tucson, AZ, 1982.

[Emtage & Deutsch 1992] A. Emtage and P. Deutsch, Archie - An Electronic Directory Service for the Internet, Froc. USENIX Winter Conf., pp. 93-110, January 1992.

[Furlani 1991] J. L. Furlani, Modules: Providing a Flexible User Environment, Froc. USENIX Large Installation Systems Administration V Conf., October 1991.

[Gifford et al. 1991] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole, Jr., Semantic File Systems, Froc. 13th ACM Symp. Operating Syst. Prin., pp. 16-25, October 1991.

[Kahle & Medlar 1991] B. Kahle and A. Medlar, An Information System for Corporate Users: Wide Area Information Servers, *ConneXions - The Interoperability Report,* 5(11), pp. 2-9, Interop, Inc., November 1991.

[Kernighan & Ritchie 1988] B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* 2nd Edition, Prentice Hall, Englewood Cliffs, NJ, 1988.

[Knuth 1984] D. E. Knuth, *The TeXbook*, Addison-Wesley, Reading, MA, 1984.

[Lamport 1986] L. Lamport, *LaTeX: A Document Prepartion System,* Addison-Wesley, Reading, MA, 1986.

[Leffler, et al. 1989] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System,* Addison-Wesley, Reading, MA, 1989.

[McCahill 1992] M. McCahill, The Internet Gopher: A Distributed Server Information System, *ConneXions - The Interoperability Report,* 6(7), pp. 10-14, Interop, Inc., July 1992.

[Muntz & Honeyman 1992] D. Muntz and P. Honeyman, Multi-Level Caching in Distributed File Systems — or — Your Cache Ain't Nuthin' But Trash, Froc. of the USENIX Winter Conf., pp. 305-313, San Francisco, CA, January 1992.

[NeXT 1991] NeXT Computer, Inc., *NeXT User's Reference,* NeXT Computer, Inc., Redwood City, CA, 1991.

[Neuman 1992] B. C. Neuman, Prospero: A Tool for Organizing Internet Resources, *Electronic Networking: Research, Applications, and Policy,* 2(1), pp. 30-37, Meckler Publications, Westport, CT, Spring 1992.

[Ousterhout et al. 1985] J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson, A Trace-Driven Analysis of the UNIX 4.2 BSD File System, Froc. 10th ACM Symp. Operating Syst. Prin., pp. 15-24, December 1985.

[Salton & McGill 1983] G. Salton and M. J. McGill, *Introduction to Modern Information Retrieval,* McGraw-Hill, New York, NY, 1983.

[Schwartz et al. 1992a] M. F. Schwartz, D. J. Ewing, and R. S. Hall, A Measurement Study of Internet File Transfer Traffic, Tech. Rep. CU-CS-571-92, University of Colorado, Boulder, CO, January 1992.

[Schwartz et al. 1992b] M. F. Schwartz, A. Emtage, B. Kahle, and B. C. Neuman, A Comparison of Internet Resource Discovery Approaches, *Computing Systems,* 5(4), University of California Press, Berkeley, CA, Fall 1992.

[Schwartz & Tsirigotis 1991] M. F. Schwartz and P. G. Tsirigotis, Experience with a Semantically Cognizant Internet White Pages Directory Tool, *J. Internetworking: Research and Experience,* 2(1), pp. 23-50, March 1991.

[USENIX 1986] USENIX Association, UNIX Supplementary Documents, 4.3 Berkeley Software Distribution, November 1986.

[WAIS 1992] WAIS server sources, available by anonymous FTP from think.com:/wais/wais-sources.tar.Z.

[Wall & Schwartz 1991] L. Wall and R. L. Schwartz, Frogramming Perl, O'Reilly and Associates, Inc., Sebastopol, CA, 1991.

Author Information

Darren R. Hardy earned a B.S. in Computer Science from the University of Colorado, Boulder, and is currently completing an M.S. in Computer Science. He specializes in network resource discovery, distributed systems, and information retrieval. As a systems engineer at XOR Network Engineering, Inc., he develops network support software and Internet utilities. Hardy can be reached by US Mail at the Computer Science Department, University of Colorado, Boulder, CO 80309-0430; or by electronic mail at hardy@cs.colorado.edu.

Michael F. Schwartz received his PhD in Computer Science from the University of Washington. He is currently an Assistant Professor of Computer Science at the University of Colorado, Boulder. His research focuses on issues raised by international networks and distributed systems, with particular focus on resource discovery and network measurement. Schwartz chairs an Internet Research Task Force Research Group on Resource Discovery and Directory Service, and is on the editorial boards for *IEEE/ACM Transactions on Networking* and for *Internet Society News*. Schwartz can be reached by US Mail at the Computer Science Department, University of Colorado, Boulder, CO 80309-0430; or by electronic mail at schwartz@cs.colorado.edu.

Hardware Profiling of Kernels

Andrew McRae – Megadata Pty Ltd.

ABSTRACT

Or: How to look under the Hood while the Engine is Running.

This paper describes a method of accurately measuring and profiling kernel code in real time with cheap and readily available hardware. Other profiling methods are touched upon, and why these methods were rejected. Some goals are stated, and a proposed hardware/software solution is described. In a case study, a 386BSD kernel is evaluated, and the results of this exercise are presented, demonstrating how tracing of software in real time highlights optimal or non-optimal code paths. The solution also provides for effective and flexible kernel debugging.

Warning to software people: this paper contains some descriptions of hardware.

Introduction

Michael Jackson has made some pertinent remarks about optimisation.

Jackson's First Rule of Optimisation:
Don't do it.

Jackson's Second Rule of Optimisation (for very experienced programmers):
Think about it, then don't do it.

This expresses a well founded caution, often ignored by the naive, who would do well to learn an important lesson:

Make it right before you make it fast.

Even so, much effort goes into making programs as fast as possible, leading to a plethora of optimising pre-processors, compilers, assemblers etc. However, with a poor design, the best optimising compilers are usually of little benefit. Experience has shown that if a piece of software is not performing, reviewing the design is the best, and sometimes only, way of obtaining significant improvement. Sometimes performance is not a major goal of software; other issues such as maintainability, correctness under all conditions, and robustness are more important. Other times it is important that a piece of software not only runs correctly, but runs fast as well. The ideal is to have the best design, and then apply optimisation so that the implementation can perform well.

It is a common mistake to expend effort optimising code that intuitively seems to be slow, but contributes only a small portion to the overall total, and not optimising where most of the time is spent. This is oft referred to as the *10/90 rule*, where if a piece of software was improved in speed by 10%, and that software contributed only 10% of the running cost, an overall gain of only 1% is obtained; if the 10% improvement were applied to software that was 90% of the running cost, then a 9% overall gain is gained.

The key to optimisation is to understand where or how it is applied (or whether it should be applied at all), and therefore the Golden Rule of Optimisation is:

Measure BEFORE you optimise.

Optimisation in UNIX

UNIX has a number of tools to help in this area; compiler profiling allows time based and function entry/exit profiling to be incorporated into programs, which allow operating statistics to be extracted and analysed. Generally this is sufficient for most programs, as the programs are not usually interacting with, or affected by, real world events. Simulators also have been used to good effect by providing a higher degree of granularity to profiling, allowing tracing of code paths etc.

Kernel Profiling

Kernels are a special case in that they must interface to real world entities, such as devices, networks, memories, clocks etc. Subtle and complex interactions occur between device drivers, processes and external events, as anyone who has attempted to remedy bugs caused by these interactions will appreciate. It is likewise difficult to obtain hard data to guide kernel optimisation, mainly due to the difficulty in obtaining fine-grained kernel performance measurements.

Kernel measurement has been considered a Black Art in the past. A number of techniques have been devised that allow various degrees of accuracy. Virtually all kernels keep event statistics and counters that allow a rough idea of the overall performance; these counters can be reset or logged at specific intervals to give a broad understanding of system activity. Examples include paging rates, network packet inputs/outputs, disc block transfers etc. The main drawback to relying on event statistics is the poor granularity and lack of detail concerning where the kernel time is spent. Keeping a large

number of statistics also takes up memory, and sometimes requires a not insignificant amount of CPU time to update them.

A more common approach is to measure the overall system performance by using an external benchmark package, or by timing the throughput or response time of the kernel by running specialist programs, e.g., *ttcp* (networking), *iozone* (file system) etc. Others run a sample of the intended applications so that a true idea is obtained of the system performance in that environment. Whilst these are the ultimate in kernel measurement (by definition), they do not aid in discovering where optimisation should be employed, except perhaps in a general sense ('the network code needs to be faster...'. 'But where in the network code?').

Some areas of kernels can be measured in the same way as user programs, using function counting and gross clock profiling. If a psuedo-random or skewed clock is available, then it is possible to improve the clock profiling so that other clock-related activity is not missed. These measurements are useful but suffer from a trade-off in granularity and accuracy; the finer the granularity, the more time is spent running the profiling clock and not actually running the kernel, which may perturb the kernel's activity. The coarser the granularity, the less effect on the kernel activity, but then the resolution becomes too low to perform useful measurement. Memory also has to be reserved to store the profiling clock data, and having clock profiling running often may cause instruction and data cacheing to be adversely affected (though with larger caches becoming more common this may not be significant).

But what happens if one wishes to profile the clock interrupt code itself? What happens if you wish to measure the time taken to process character input interrupts, or discover the optimal code path taken for processing back-to-back packets through a certain protocol stack, checking the time to reply with acknowledgements?

The fly in the ointment is that kernel profiling is like the Heisenberg Uncertainty Principle i.e the more accurate your measurements, the more you are perturbing the environment in which the kernel is running, and the less likelihood of getting data which reflects the actual state of the unprofiled kernel.

Other methods are available which are non-intrusive, such as connecting large amounts of hardware to record the instruction stream; this is expensive and requires specialised hardware, normally out of the league of the casual kernel hacker. Another problem with this method is that it often does not cope with cache effects; instruction caches must be turned off, thus ruining the non-intrusive nature of the measurement. Microprocessor designers are becoming aware of the need to measure and trace processor activity even when running in cache, and newer designs often have pins dedicated to providing indication of the state of the processor.

So we are faced with a dilemma; in order to rationally test kernel designs and code, we need accurate measurements, but in obtaining these measurements we change the environment of the kernel, and possibly introduce erroneous measurands (and consequently make wrong design decisions). Any kernel profiling system must be as non-intrusive as possible, or at least keep the effect of measurement to a minimum so that it does not grossly alter the timing characteristics.

The Goals

As a result of much software written in an embedded environment, a great deal of it driver and kernel related, I became increasingly interested in being able to easily measure and profile the software, and so make rational and informed judgements concerning algorithms and coding techniques. Faced with the regular need to discover why things were not responding at the expected speed, it quickly became clear that the human brain is not a good enough simulator to handle the complex timing interactions occurring within a kernel. Some early solutions to the problem was to use statistic counters, but this was usually too gross a measurement to help. Another favourite method was to press-gang a hardware engineer to connect an oscilloscope to the equipment; this allowed external responses to be measured, and certainly helped when hardware drivers were being tested.

Sophisticated tools such as logic analysers provided a major benefit, as whole sequences of events could be trapped and examined in the cold light of day. More intelligent software within the analysers allowed instruction disassembly, which made easier work of following code paths, but this was generally tedious and unfriendly because of the difficulty in relating the raw instruction stream back to the source code. It also is not trivial to connect and operate a logic analyser for most software engineers. Special logic analyser software can be used to perform time based profiling, but the sampling granularity was generally too coarse to be of any use, and since it operated on physical addresses, this was difficult to relate back to the actual software.

In-circuit Emulators generally are considered the top of the heap for embedded development, and come with complete suites of cross-compilers, assemblers, remote debuggers and hardware which allows all manner of tracing and measuring programs. They also come with Rolls Royce price tags. Unfortunately they tend to be black boxes when it comes to analysing the data; it is often difficult to extract the desired information from the raw timing data, and then integrate the information with the source code.

I still had a desire to find out what was really happening *inside* these kernels, but I had a limited budget. I wanted to do the equivalent of what our local car mechanic does, to open the hood, listen to the engine running, judge the revolutions, feel the temperature, and so forth.

By now I had attempted several methods of getting the data, with limited success, but in the meantime I had formulated a wish list to describe what I wanted.

- Fine granularity of measurement, so that accurate profiling may be obtained.
- Little or no intrusiveness, so that performing the measurement will not affect the timing of the kernel.
- Integration with development tools or program source so that source level code paths may be traced with ease.
- Profiling to occur for all kernel operations within a selected interval, including clock interrupts, device interrupts, even sections when processor interrupts were locked out.
- If some hardware assists were to be employed, then some easy and portable method of connection should be used, e.g., not having to connect 96 separate clips to a PCB.
- Immune to instruction cache effects. In fact it should still work as expected with instruction cacheing enabled (as any 'production' code would run the cache enabled).
- Granularity to a source code function level (however short the function is) should be the worst case; however it would be desirable to profile within functions if possible.
- Any method of profiling should be portable between different computer architectures.

It became clear that it is impossible to fulfill these goals with software alone. It is also clear that complex hardware did not offer an elegant (or cheap) alternative. This paper describes a solution to this problem which is a better alternative to software only kernel profiling, and much cheaper than specialised and complex ICE hardware measurements of kernel operation. Cheap enough that any person who wished to profile and debug their home PC would be able to put it together, but useful enough so that design decisions could be made in confidence as a result of accurate measurement. It attempts to meet the above goals, and also be simple and cheap enough to build without great effort (even a software engineer could probably manage it).

The Profiler

Three basic elements are used in the profiling system proposed; the first is a hardware device that is used to record time and event data into a RAM block. The second is a modified C compiler that allows event triggering code to be inserted into key locations, and finally the last building block is analysis software that is used to decode the back-trace of events and relate it to the source code.

The Hardware

The role of the hardware in the Profiler is very simple. Its job is to store timing information and some identification value. It is purposely as simple as possible, primarily because it was a first attempt at exploring what the basic hardware requirements were for meeting the goals. A lesser goal was cost minimisation; as long as the cost could be held to something below one or two hundred dollars than the Profiler could be built by just about anybody.

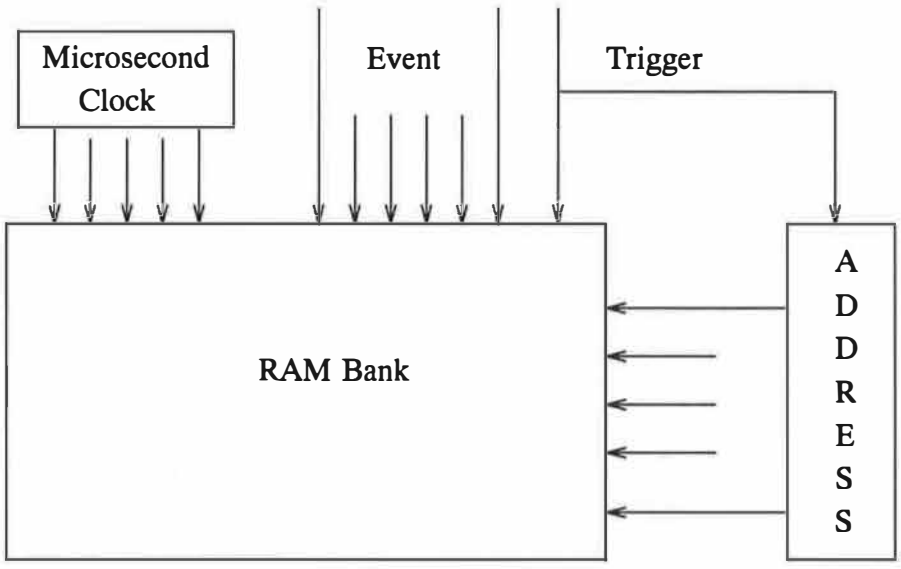


Figure 1: Profiler block diagram

Finally the Profiler is simple because I hate wire wrapping; it's so much more tedious than writing software.

Commonly available components were used, and the hardware prototyped on a breadboard using wire wrapping. A single electrically erasable PAL is used for the logic and timing functions; the final cost of the parts totaled less than $100 dollars. It has a chip count of 5 static RAMs, 5 counters, 1 PAL, 1 oscillator and 1 delay line. Having an EE PAL turned out to be a great boon, as it meant quite a bit of experimenting could take place to get the logic right, and also meant that moving to equipment that used different methods of accessing the Profiler could be handled by different PAL equations. It also allowed extra facilities to be incorporated such as some display LEDs and control switches.

A block diagram appears in Figure 1. The Profiler consists of a block of RAM which is 40 bits wide, an incrementing address counter, a free running counter clocking at 1 Megahertz, and some control logic. The RAM is split into two sections, one holding an identification code (**event tag**) which is 16 bits in width, and the other 24 bit wide section connected to the microsecond clock. When an event tag is presented to the Profiler, it stores this code along with the microsecond counter value into RAM. The RAM address is automatically incremented every time an event is stored, essentially storing the event and time in a large list. The list is currently 16384 events long, but there is no inherent limit to the total number of events stored except the maximum amount of memory designed into the Profiler.

The microsecond timer is 24 bits long, allowing a maximum time of 16 seconds between events before the time is wrapped around and information is lost. Note that this is the maximum time between events, not the total time that can be profiled - the analysis software only uses the timer value as an interval time, not as an absolute time. The event tag is 16 bits, allowing 65536 unique event tags.

The trick in this scheme is not the gathering or storing of the event/time data (a Simple Matter Of Hardware), but how to generate the event code, which must come from the equipment being measured. It was clear that some software assist was required to generate these event tags in an orderly fashion. Another problem was how to connect the Profiler to a working system.

An elegant solution presented itself when I realised that most of the computing equipment that the Profiler was designed for has one or more EPROM sockets fitted for boot code or board drivers. This presented itself as a simple method of connecting the Profiler to the equipment, by piggybacking a EPROM socket onto some cable, and using the socket to bring the appropriate signals into the Profiler. The original boot EPROM would plug into the piggy-back socket, if indeed it was required. The event trigger would be the access of the EPROM, and the address of the EPROM access could be the event tag data.

In this case, only 18 signal lines needed to be brought into the Profiler (16 address lines and the EPROM ChipEnable and OutputEnable signals). This allowed a simple and easy method for the Profiler to connect to **any** piece of equipment that contained a standard EPROM socket, without other connections. Power is obtained from the EPROM socket, so the Profiler is self contained.

A switch exists on the Profiler that initiates the profiling recording; this allows the Profiler to be synchronised with execution of test programs, network activity etc. Two LEDs exists in the card giving some indication of its state; the first indicates that the Profiler is active and storing data, the second indicates that the address counter has overflowed and the Profiler has automatically ceased storing data.

The profiling scenario is now clear; simple software triggers are sprinkled in strategic locations throughout the target software. Each time one of the triggers is executed the time and trigger value is recorded. How does the data then get retrieved? The data RAMs are mounted via battery-backed SmartSockets™, and when the profiling samples are stored, the timing data is retrieved by transferring the RAMs into another networked embedded host, and copying the profile data to a UNIX host for processing.

And so I had a workable hardware/software scheme that could record with accuracy specific events occurring, was easy to connect to a piece of equipment, didn't require a lot of signal hooks, and the software trigger was minimal enough not to intrude very much in the timing of the kernel.

Generating the Triggers

The next problem was how to manage the event triggers i.e how to automatically generate them in the target code, and how to generate the event value so that it could relate back to functions and points within functions.

It seemed natural to place a trigger at the entry and exit of each function; in this manner code paths could be traced, and accumulated times calculated for each subroutine. It isn't really practicable to modify the source code to explicitly add the triggers; this would mean that a macro would have to be used so that the profiling could be turned off, and it would also mean manual allocation of a trigger value to each function, something that is tedious and error prone. Besides, many functions have multiple exit points, and often functions contain some initialisation as part of their local variable declarations which would be performed before the trigger; this would give skewed timing results.

So it was decided to modify the compiler to add the trigger points; the Free Software Foundation's GNU C compiler was modified to generate the triggers at the start and end of every function. For ease of processing and identification, each function is assigned a trigger value that is an even number, and that number + 1 is used as the function exit trigger. On a 68000 system, this effectively added one instruction in the function prologue, and one instruction in the function epilogue, e.g., a function would now contain:

```
.globl _myfunction
_myfunction:
    tstb 1386
    link a6,#-8
       ...
    unlk a6
    tstb 1387
    rts
```

If a higher granularity of profiling is required with a function, then a macro may be used to generate an *inline* trigger via a compiler *asm* function. Assembler routines may have event tag trigger instructions added via an include file and a preprocessor macro.

The trigger value is taken from a file containing the function names and values, of which a sample is shown below:

```
main/502
hardclock/510
gatherstats/512
softclock/514
timeout/516
untimeout/518
swtch/600!
MGET/1002=
```

The insertion of Profiler event tag instructions is enabled by a compiler option indicating the name of the file containing the functions names and event tag values. This file is automatically extended by the compiler when it generates new event tags for functions that do not already exist in the file; the event tag for the added functions is taken as the next available value (i.e the next value higher than the current highest in the file). The name/event tag file may be generated from scratch, with an initial dummy entry indicating the starting tag number to use. Once generated, the same profile tags are used to allow recompilation without having different profile tags assigned to a function. Multiple name/tag files may exist, and may be concatenated to provide a complete list of profiled functions. Inline and assembler trigger names and values may be manually added to the file.

Special character modifiers may be appended to any of the name/tag values that indicate special processing of this particular tag when analysing the results; a '!' character indicates a function that causes a processor context switch, which the analysing software must treat specially. The '=' modifier indicates an inline tag, as opposed to a tag representing the entry or exit of a function.

Adding event tag triggers to software will have a small impact on performance; this has been calculated at around 1 to 1.2% extra CPU cycles, which is a small penalty to pay for profiling. In absolute terms this equates to about 400 nanoseconds per function for a 40 MHz 386. The size of the software also increases by the overhead of two instructions per function; it is hard to quantify this increase as a percentage as it depends on the number and size of each function, and also on the number of inline triggers used.

Connecting to a PC

The initial platform for testing was a 68020 board designed for embedded applications. Since it was of Megadata design and manufacture, it was

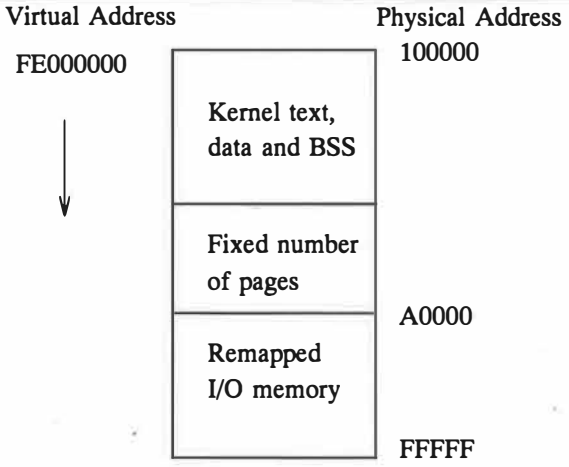


Figure 2: Virtual memory remapping

easy and safe to develop and test the Profiler hardware on this platform.

Once the concept was tried and proven, it was decided to connect the Profiler to a *real* kernel, namely the freely available 386BSD release 0.1 running on a 40 Megahertz 386 PC with 8 Megabytes of memory. Since the point of interface was a common JEDEC EPROM socket, it was simple to connect the Profiler to the PC via a spare EPROM socket on a Western Digital WD8003E Ethernet controller. Any ROM socket could have been used as long as it was at a known fixed address and was accessed as a 8 bit wide device, such a VGA BIOS ROM socket etc. The address space of the ROM falls somewhere in the ISA bus memory address space, between (hex) A0000 and 100000.

Changes were made to the 386BSD C compiler (based on gcc 1.39) to accommodate the Profiler event tag additions. A snag was hit when it was realised that the 386BSD kernel remapped the kernel's view of ISA bus memory into kernel virtual address space, and so an absolute address could not be easily used.

After initial loading, the 386BSD kernel remaps the physical memory addressing to new virtual locations as shown in Figure 2.

In effect, the kernel is remapped to absolute location FE000000; the last location of the kernel is rounded to a page boundary, and a fixed number of pages are allocated for the kernel stack, a proto u-dot area and other virtual memory requirements. The ISA memory address space is then remapped to follow this kernel address space; the virtual address that this memory is mapped at may vary depending on the size of the kernel.

The Profiler event tag instructions added by the compiler require an absolute address within the EPROM address range starting at a fixed EPROM location somewhere in the ISA bus memory address space. But since this EPROM location may vary depending on the kernel size, it cannot be resolved at compile time. It would be unreasonable to have to recompile all source code modules just to update the event tag instructions. Fortunately it can be resolved at link time with a little extra effort; the compiler modifications generate function entry and exit event tag instructions thus:

```
.globl _myfunction
_myfunction:
    movb _ProfileBase+1386,%al
    pushl %ebp
    movl %esp,%ebp
    subl $8,%esp
    ...
    leave
    movb _ProfileBase+1387,%cl
    ret
```

The global label *_ProfileBase* is set in an assembler file as a result of a two stage kernel linking process. The kernel is first linked with a dummy of _ProfileBase, then a shell script is automatically used to extract the size from the kernel and recompile the assembler file with the real value of _ProfileBase, which is then linked with the kernel. If the physical address of the Profiler EPROM location is changed, then only this assembler file has to be modified to cater for the new position of the EPROM. This scheme worked very well in providing a correct run time virtual address of the Profiler's physical memory address.

Profiling the Kernel

A total of 16384 event tags and time values may be stored in the Profiler before the RAM addressing overflows. Whilst this allows a considerable amount of data to be gathered, if a particular subsection of the kernel was to be examined in finer detail, then some form of selective profiling should take place. This is easy to set up, as all that needed to take place was to compile those modules of interest with profiling enabled, and to compile the rest of the kernel without profiling. This allowed highly selective profiling to take place without losing resolution, but without filling the Profiler RAM with events in which there was no interest.

This selective profiling allowed two broad categories of profiling to take place, macro-profiling and micro-profiling. Macro-profiling takes place when certain key modules such as the system call handlers and VNODE interface routines are profiled. Virtually all kernel code paths traverse these higher level routines, so it is possible to get a broad-brush view of system performance to answer questions like, "How long does it take to fork/exec a process?" Or "How long does it take to read this file?" Or "How long does it take to open a TCP connection?" This view of the kernel is very instructive as the overall code path through the kernel can be easily seen and traced, and can give a guide to where further profiling should take place.

Micro-profiling takes place when a particular subset of the kernel is examined in detail. Interrupt handlers, clock routines, assembler subroutines can be profiled as well, allowing complete snapshots to be taken of a particular kernel code path. For example, the file system buffer cache, file system code and disk driver routines can be profiled, so that whenever the kernel enters these areas, the code path is traced. No other code paths are profiled, allowing a detailed and unobstructed view of that section. Similar subgroupings may be made with the networking code, the Network File System (NFS) code, the virtual memory subsystem, various drivers (SCSI, tty, IDE) etc.

After repeated micro-profiling of the various kernel subsystems, it is possible to eventually

construct a highly detailed and accurate mosaic of the kernel performance. As a result, quantitative comparison may guide design and implementation improvements as performance bottlenecks are highlighted in the kernel, and accurate before and after measurements may be made to test the success of such changes.

Analysing the data

Once the triggers are generated in the object code, and the Profiler has captured some events, the raw data is then uploaded to a UNIX host. The data is processed by matching the event data (with the microsecond time values) with the function names as listed in the name file. The raw data appears as a list of event tags and times. How then is the data processed to gain the maximum useful information out of it?

Identification of function entry and exit points allow a code path trace to be constructed with timing information at each call and return point. Subroutine depth is easily discovered by matching exits with entries, with event tags between a function's entry and exit indicating subroutine calls within that function.

This works well when used when the control flow follows a simple subroutine call/return model, but when the target being profiled is a kernel this model is inadequate to describe the thread of control. The essential difference is that the kernel is multiplexing many processes, and context switches occur to change the control flow to a different process. This appears in the profiling data as a discontinuous change in the subroutine call/return model, where it appears a different subroutine is being exited than was called. Some extra information must be given to the analysing software to indicate where context switches may occur.

386BSD context switches occur in the *swtch()* function; upon entry to *swtch* the current process context is saved, and the run queue is checked for the next process to run. If none are ready, then an idle loop is entered.

The analysis software must detect when *swtch* is entered so that each process's code path may be analysed separately. The *swtch* function is tagged in the name file with a modifier to indicate this special processing. The time between the exit of a call to *swtch* and the entry to the next call of *swtch* is analysed as a contiguous block of processor activity. The time in *swtch* itself is counted as CPU idle time, except when device interrupts occur. The separation of idle and active CPU time provides accurate calculation of CPU usage, both as a overall ratio and on a per function basis.

Currently two different analyses can be generated; the first is a summary of each function's statistics, sorted by highest to lowest net CPU usage, headed by an overall summary of the profiling data, see Figure 3.

The elapsed time for each function is the accumulated interval time recorded between the function entry and exit. The net time is the accumulated time minus the accumulated time of all subroutines that are called from this function, giving an overall time for this function alone. The count of calls to each function is calculated, as well as the maximum, minimum and average time spent in each function. The net time is expressed as a percentage of the absolute elapsed time for the entire run (*% real*), and also as a percentage of the total time the processor was not sitting in the idle loop (*% net*).

These statistics give accurate and concise summaries of the processor activity, and can quickly highlight bottlenecks or subroutines that are heavily used. As can be seen in the example, it is obvious

```
Elapsed time = 0 sec 497272 us (28060 tags)
Accumulated run time = 0 sec 492248 us (98.99%)
Idle time = 0 sec 5024 us ( 1.01%)
-----------------------------------------------
  Elapsed  Net   # calls  (max/avg/min)  % real  % net
  166218 165343     889    (1089/185/2)  33.25% 33.59%   bcopy
  152382 151700     514    (901/295/23)  30.51% 30.82%   in_cksum
   26359  26359    2474       (13/10/8)   5.30%  5.35%   splnet
  442031  16391     166     (125/98/87)   3.30%  3.33%   soreceive
    9963   9913    2782        (19/3/3)   1.99%  2.01%   splx
   16069   9855     433      (36/22/18)   1.98%  2.00%   malloc
  202651   9132      86    (193/106/28)   1.84%  1.86%   werint
  183830   7989     170      (98/46/18)   1.61%  1.62%   weget
   13646   7576     423      (23/17/15)   1.52%  1.54%   free
   19467   7189     218      (78/32/12)   1.45%  1.46%   westart
   ...
```

Figure 3: Summary of profiling data

immediately that the CPU is completely saturated, and most of its time is spent in *bcopy*.

The second report shows a real time code path trace, along with accumulated and separate function timings. Subroutines are shown as nested where necessary to allow easy following of the code path; a sample is shown below in Figure 4.

Accumulated and net elapsed times are shown for each function, e.g., the *tcp_input* function takes 318 microseconds total elapsed time, but only 92 microseconds was actually spent in the *tcp_input* routine; the other 226 microseconds were spent in subroutines called from within *tcp_input*. Inline triggers are marked using '=='. Modifiers in the names file allow detection of context switches, which are flagged in the code path trace.

From the function summary report *bcopy* is a likely target for more investigation; each invocation of bcopy can be examined by looking at the code path trace, and some idea can be obtained why this function is causing high CPU usage.

Much of the effort going into the Profiler now centres upon processing the raw data in many more useful ways, such as graphically representing the code path or building histograms of the function time and usage for easy detection of bottlenecks.

User Code Profiling

The hardware profiling solution can be readily adopted to user level profiling with similar results. A driver stub may be configured in the kernel that reserves the Profiler's physical memory address space; a modified profiling *crt.o* initialises the process for profiling by opening the driver and calling *mmap* to memory map the Profiler's address space into a fixed location within the process address space.

There is no reason why a mixture of kernel and user level profiling cannot take place concurrently, or profiling several user processes at the same time to closely monitor and analyse interactions occurring via the interprocess communications facilities. This approach is especially applicable in debugging and tuning communication protocol stacks, where the network and link layers are implemented in the kernel, and the transport layer and higher layers are implemented in user libraries and application code.

Case Studies

The first platform that the profiler was tried on was a 68020 based embedded system running a Megadata kernel incorporating the 4.3 BSD Tahoe release networking code. A number of profiling

```
0:002 671 -> ISAINTR (31 us, 778 total)
0:002 679     -> weintr (50 us, 292 total)
0:002 704         -> werint (70 us, 215 total)
0:002 739             -> weread (11 us, 145 total)
              ...
0:003 458                     -> bcopy (1073 us)
              ...
0:004 996     -> ipintr (55 us, 424 total)
0:004 998         -> splnet (10 us)
0:005 012         -> splx (4 us)
0:005 031         -> in_cksum (23 us)
0:005 074         -> tcp_input (92 us, 318 total)
0:005 082             -> in_cksum (38 us)
0:005 138             -> in_pcblookup (9 us)
              ...
0:005 424     -> spl0 (21 us)
0:005 449 <-
            ---- Context switch in ----
0:005 488                 <- swtch
0:005 492                 -> splx (3 us)
0:005 513             <- tsleep (22 us, 25 total)
0:005 520             -> falloc (22 us, 83 total)
0:005 523                 -> fdalloc (13 us, 18 total)
0:005 528                     -> min (5 us)
0:005 541                 <-
0:005 547                 -> malloc (29 us, 43 total)
              ...
```

Figure 4: Code path traces

studies helped greatly in identifying key performance problem areas in the kernel, and in one case the recoding of an Ethernet driver doubled the network throughput.

A SNMP client based on the CMU SNMP code was profiled, highlighting a major bottleneck in searching the MIB table linearly; redesigning the data structure to use a B-tree to hold the MIB data reduced the CPU cycles required to respond to SNMP requests by an order of magnitude.

Since the embedded system contained no memory management hardware, no user/kernel boundary existed except as an artifice of the system interface, thus it was easy to trace activity right from application level code down to the kernel code through to driver code.

The next step was to begin profiling the 386BSD kernel, which provided much more comprehensive and interesting results. These results are presented in several sections, the first being an overall impression of performance, and the other sections taking one kernel subsystem and describing the results of profiling each one in turn.

386BSD Overall Performance

The profiled kernel contains 1392 functions, so 2784 event tag trigger points were automatically added to the code. 35 assembler routines had trigger points added, so a total of 1427 possible functions could be profiled. Depending on the nature of kernel activity, the Profiler RAM could be filled (a total of 16384 events) in as short a time as 300 milliseconds. No noticeable difference can be detected between a profiled and a non-profiled kernel. After profiling a number of the key areas of the kernel, some impressions emerged concerning the kernel performance. These fall into three main categories; CPU performance, I/O performance and virtual memory management.

Firstly, I was pleasantly surprised to note the oft maligned Intel architecture did indeed run fast, especially at a clock speed at 40 Megahertz and employing 64 KB of external cache. Moving data through the kernel to user space was faster than expected, and it was clear that function call and return was also speedy. It would be instructive to profile other microprocessor types running at a similar speed using the same software to do a side-by-side comparison. Undoubtedly memory speed and cache effects have a major impact on performance, as data throughput dropped markedly whenever memory was accessed on the ISA bus as opposed to main memory. More on this later. Profiling the interrupt code showed that the regular clock tick interrupt took on average 94 microseconds to execute; unfortunately the hardware architecture does not provide for Asynchronous System Traps (commonly known as software interrupts), so the interrupt code has to work extra hard to emulate this facility. The interrupt code overhead to do this is around 24 microseconds per interrupt; it is hard to judge whether this has a significant impact on system performance.

Due to the interrupt architecture of the bus and the processor, it was evident that more time was spent ensuring correct synchronisation and interrupt lockouts than would normally be required on a multi-priority interrupt level processor such as 680x0; on the average it took 11 microseconds per *splnet* call, which may not seem a long time, but the *spl** routines get called a great deal, and it all adds up to a significant amount. In one test, 9% of the total CPU time was spent in *splnet, splx, splhigh* and *spl0*. Unfortunately it is hard to see how this could be improved, given the nature of the interrupt architecture.

Some sample functions are shown in Table 1, along with their measured average execution times (the times are inclusive of subroutines that are called).

Function	Microseconds
vm_fault	410
kmem_alloc	801
malloc	37
free	32
splnet	11
spl0	25
copyinstr	170

Table 1: Sample function timings

When some tests were performed where input/output activity was heavy, it was clear that a major bottleneck in system performance is the use of the ISA bus. This was especially noticeable on the Ethernet adaptor, which is a 8 bit wide controller. To transfer similar amounts of data, the ISA bus is up to 20 times slower than main memory transfers.

It would be instructive to profile different controller cards to determine where each performed best; when support for EISA cards is available it would be interesting to see what performance gain would be obtained using the higher bandwidth bus.

Whilst the CPU performs reasonably well, overall performance is crippled by the poor I/O bandwidth, and the interrupt architecture of the 386 and the ISA bus also contributes to reduced performance.

The virtual memory management subsystem of 386BSD was derived from the Mach memory management code; a member of the CRSG has been heard to say that the old BSD VM code was ripped from the kernel, and the Mach memory management code placed next to the kernel and hot glue poured down the middle. Following code path traces of various virtual memory functions seem to support this

model, and it seems the glue is fairly thick in some places and thin in others. Some functions seem to run surprisingly fast; the routine that handles page faults and enables new pages to be accessed (*vm_fault*) takes about 400 microseconds, which seems reasonably low overhead. On the other hand, an excessive number of page faults seem to occur at times. Where the real performance problems lie is in creating new VM contexts for new processes, as explained in the next section.

Fork/exec Profiling

A common operation of UNIX is to *fork* a process and create a child copy of the process, which then *execs* a new process image. For UNIX to perform well, these two operations must be reasonably fast, since some UNIX operations rely on a low cost of process creation.

The current situation looks fairly abysmal; it takes some 24 milliseconds to perform a *vfork* operation, and it takes about 28 milliseconds to perform an *execve* system call. This adds to about 52 milliseconds to perform a combined fork/exec operation. Note that these times do not include any disk activity, as the process image was already cached. Where is this time being used? In figure 5 a summary of the highest cost subroutines is shown.

Most of the CPU time occurs within a small number of routines; it is clear that the *pmap* module is a bottleneck when manipulation of the virtual memory is required (the *bcopyb* call relates to scrolling of the console screen, so it should be ignored for the purpose of the exercise). Over 50% of the time is being spent in the virtual memory routines shown above. Examination of the code path trace shows that *pmap_pte* is called 1053 times when a *fork* is executed, and a similar amount when an *exec* is done. Further analysis of the code path shows the exact progress of the fork operation, and each subsection can be examined in detail to see the amount of time it is taking, and whether significant optimisation can take place. There is a major amount of cross-calling between the *pmap* module, and the rest of the virtual memory subsystem, so it is envisaged that a major performance benefit would occur if some of that glue could be trimmed back and some sculpting of the interface performed.

Network Performance

Profiling was performed on the TCP/IP and socket code by running a program that listened on a socket and when another host connected, read and discard the data. A Sun Sparcstation 2 was used as the host to send the data, as I was sure it could fill the available network bandwidth to the PC over an ethernet.

This was the only test that caused the PC to be totally CPU bound, so that essentially the CPU was busy 100% of the time. It was obvious that the PC could not process the data from the network at anywhere near Ethernet speed. Examining the code path trace and function summary showed that 33.6% of the time was spent in *bcopy*, and that 30.8% of the time was spent in *in_cksum*. Again, *splnet*, *splx* and *spl0* contributed around 9% of the time.

Delving further into the code path trace, it was clear that a major bottleneck occurs because the Ethernet driver for the card must copy that data from the onboard controller memory across the bus; each TCP data packet that was received (i.e a full Ethernet packet) took about 1045 microseconds to process at the driver level. This alone is only about 20% more data throughput than Ethernet itself, so it is unlikely that Ethernet data rates through to the network applications can be achieved using this 8 bit controller card, unless the rest of the software has been tuned for minimum overhead. One approach to solve this copying overhead is to make the buffers on the controller memory external *mbuf* memory, so that all the driver has to do is link the received packet(s) to *mbuf* headers, and then the double copying would be avoided (once from the controller

Elapsed	Net	# calls	(max/avg/min)	% real	% net	name
77603	58913	67	(14061/879/2)	5.02%	28.22%	pmap_remove
22283	22148	5549	(66/3/2)	1.89%	10.61%	pmap_pte
12938	12938	1215	(13/10/9)	1.10%	6.20%	splnet
10912	10874	3	(3634/3624/3613)	0.93%	5.21%	bcopyb
33435	10134	453	(40/22/21)	0.86%	4.85%	spl0
15963	7876	8	(3862/984/3)	0.67%	3.77%	pmap_protect
5657	5657	77	(244/73/3)	0.48%	2.71%	bcopy
47723	4889	115	(64/42/27)	0.42%	2.34%	vm_fault
4759	4759	1349	(5/3/3)	0.41%	2.28%	splx
7836	4361	236	(29/18/13)	0.37%	2.09%	vm_page_lookup
7320	3489	119	(39/29/12)	0.30%	1.67%	pmap_enter
3457	3457	38	(132/90/2)	0.29%	1.66%	bzero

Figure 5: High cost subroutines

memory to mbufs, and then via *copyout* to the user data space).

Would this help? Contrary to intuition, this would actually decrease the performance, and using the accurate timing provided by the Profiler, a close estimate of the impact can be calculated. It takes *bcopy* around 1045 microseconds to copy 1500 bytes from the controller; *copyout* takes about 40 microseconds to copy a 1Kbyte mbuf cluster to the user data space. If the controller memory were accessed only once, then collapsing *bcopy* and *copyout* would give at most a gain of 60 microseconds (less than 6%). But other routines access the network packet as well, such as the TCP and IP input processing routines, and most importantly the IP checksum routine. Checksumming the packet whilst in the controller's memory would add at least an extra 980 microseconds to the overall processing of the packet. The time to process a packet would increase from 2000 microseconds to around 3000 microseconds, a big loss. It is now obvious that if you have slow controller memory, it is a big win to get it out of that memory as soon as possible into faster main store.

The other major CPU user was the checksum routine itself, which was almost a big an overhead as the driver packet copy. This was surprising at first, as the packet was now in main memory, and the checksumming should be close to memory-to-memory copying speeds. To checksum a 1 Kbyte packet was taking 843 microseconds. It was discovered that the *in_cksum* routine has not been optimally coded (e.g., like other architectures where it is done in assembler), and recoding this routine should provide a reduction in packet processing from 2000 microseconds to perhaps 1200 microseconds; this would provide a major improvement in network performance, and the limiting factor would become the memory bandwidth available to the network controller across the ISA bus.

Another conclusion that can drawn is that a much faster I/O architecture is required before serious data throughput can be expected, but I think we all knew that.

Filesystems

Separate profiling studies have been performed on the BSD Fast File System (FFS) code and the Network File System code. Due to the network performance problems discussed in the previous section, any performance issues in the actual NFS implementation are totally swamped by the I/O bandwidth limitations. An interesting situation arises due to the fact that UDP checksums are usually turned off with NFS; since the checksum routine contributed a large proportion to the CPU overhead, NFS actually provides less overhead and better throughput than an FTP style connection!

Given the tracing capabilities of the Profiler, it was easy to get accurate measurements of the network turn around time with NFS RPC calls, and to see how long to formulate the request, send it and then how long to process the reply.

The disc controller used in the target PC was an IDE controller on a Seagate ST3144 disc. The FFS profiling showed how disc seek times impact the I/O throughput. Each read of the disc varied from 18 milliseconds up to 26 milliseconds. Each write interrupt took about 200 microseconds in total, with about 149 microseconds of that being actual transfer time of the data to the controller. Interrupts seemed to be close together most of the time (< 100 microseconds), so the disc driver may well be improved by waiting a short time after transferring the data to see if the controller is ready to accept another block straight away.

Overall, the CPU was only busy for 28% of the time when doing a large number of writes, so the disc seek times are still the major influence in determining disc throughput. It was interesting to see that out of that 28%, at least 6% was spent in the *spl** routines. It would be interesting to use a different type of controller (maybe one with DMA) and see what difference it makes.

Conclusions and Future Work

The major conclusion about the performance of 386BSD is that there are a small number of areas that need addressing, that when fixed should improve the performance considerably. The hardest area to address is the virtual memory subsystem. The easiest area would the IP checksumming. The grossest area of mismatch between the hardware architecture and UNIX is the interrupt priority control and lack of software interrupts.

It was also clear that the hardware I/O performance is a major factor, and that the platform the profiling was performed on is crippled in I/O bandwidth.

Even in its simple prototype form, the Profiler has proved to be an invaluable tool for looking under the hood while the engine is running. One clumsy aspect that remains is the uploading of the Profiler data to a host for processing; currently this is manually performed, which slows down the profiling process somewhat. I am considering a new improved Profiler hardware design with more memory and some extra facilities. A higher clock precision has been considered, especially if the Profiler were connected to a upmarket workstation architecture such as a Sun or DEC; this would entail fitting a wider RAM module for accepting more clock data bits. It is unclear at this stage whether a higher clock rate is really needed, though.

The method of connection via EPROM socket has proved to be so useful that it is hard to see how

it could be improved. The next step is to bring in the EPROM data lines as well, and have a Zero Insertion Force socket for the EPROM on the Profiler itself. Then once the Profiler has been used to collect the data, each of the storage RAMs in turn can be multiplexed into the EPROM address space, and the data can be read as if it were an EPROM. This would allow fast turnaround for processing the Profiler data.

Since the raw data comes in a simple package, a lot of analysis can be applied to the raw data. Further work in this area hopefully will yield sophisticated tools that allow statistical processing of the data, groupings of functions into separate subsystems, and other ways to process the data.

Acknowledgements

Thanks must go to Bryan Grayson, who slaved with me over a hot logic analyser. Without his invaluable assistance the Profiler would still just be an idea.

Thanks also to Megadata, who suffer me being distracted from doing *real* work for long enough to try out new ideas.

Finally thanks must go to William Jolitz and the UCB Computer Science Research Group, who have brought dreams into reality for a lot of people who have always wanted to hack on kernels.

Author Information

Andrew McRae is a software engineer with the Australian company Megadata Pty Ltd, where for the last 9 years he has worked on real time supervisory and control systems. His responsibilities include communications and embedded systems, and a range of other areas such as Unix applications and drivers. He has been involved with the Australian Open System Users Group (AUUG) since 1986. Prior to joining Megadata he co-founded a company specialising in motion control special effects and computer graphics for film and television. He can be reached via electronic mail at andrew@megadata.mega.oz.au, or via snail mail at Megadata, PO Box 1687, Macquarie Centre, NSW 2113, Australia.

A Randomized Sampling Clock for CPU Utilization Estimation and Code Profiling

Steven McCanne & Chris Torek – Lawrence Berkeley Laboratory

ABSTRACT

The UNIX *rusage* statistics are well known to be highly inaccurate measurements of CPU utilization. We have observed errors in real applications as large as 80%, and we show how to construct an adversary process that can use an arbitrary amount of the CPU without being charged. We demonstrate that these inaccuracies result from aliasing effects between the periodic system clock and periodic process behavior. Process behavior cannot be changed but periodic sampling can. To eliminate aliasing, we have introduced a randomized, aperiodic sampling clock into the 4.4BSD kernel. Our measurements show that this randomization has completely removed the systematic errors.

Introduction

Traditional implementations of the Unix operating system provide coarse grained, statistical measurements of CPU utilization. On each tick of the system clock, the CPU state is examined. If the processor is in *user mode*, the current process is charged with one sampling interval of *user time*. Similarly, if the processor is in *system mode*, the current process is charged *system time*.

This approach is problematic. A process can become synchronized with the sampling clock, resulting in large scale errors in the utilization statistic. For instance, a process that runs in phase with the system clock might always surrender the CPU before the clock interrupt arrives, thereby accumulating no usage time.

CPU time estimation is of particular importance, as it drives the scheduling algorithm. If the utilization estimate is in error, scheduling, and hence system performance, can be adversely affected. Furthermore, the accuracies of the *getrusage* system call and the */bin/time* command will be compromised.

In this paper[1], we outline the theory behind the statistical CPU estimator. We then introduce a new approach based on randomization. Next, we explain how the new model fits into the current 4.4BSD system, and how it can drive code profiling as well. Finally, we give some case studies that demonstrate problems with the existing system and show that our approach has overcome them.

A Statistical Model

An exact measurement of CPU utilization would require the precise timing of every interrupt and system call. Since this is prohibitive, systems rely on a cheaper methodology based on sampling. Here, a sequence of samples of the CPU state is used to estimate the true utilization percentage, which in turn can be viewed a probability. For example, the probability that the CPU is in a given state is simply ratio of the time spent in that state to the elapsed time.

For the CPU estimator, there are three relevant CPU states: user mode, system mode, and interrupt mode. Call the probabilities of being in each of these states p_u, p_s, and p_i respectively. Then, if a process runs for T_e time units, the amount of time spent in each CPU state is simply:

$$T_u = p_u\, T_e$$

$$T_s = p_s\, T_e$$

$$T_i = p_i\, T_e$$

We need to devise a sampling experiment that produces unbiased estimates for p_u, p_s, and p_i. Moreover, the estimates should get better as we make more observations.

The observations of CPU state can be related to the probability estimates using elementary probability theory. The sequence of observations comprises what probability theory calls a *random sample*, and the Law of Large Numbers tells us that the sample mean converges to the mean of each observation, provided the observations are independent. We can view each sample as a *Bernoulli* random variable, which is 1 with probability p and 0 otherwise; its mean is p. Thus, assuming independence, the sample mean converges to p, which is what we want.

For example, consider the sequence of observations $\{U_1, U_2, \ldots, U_n\}$, where U_k is 1 if the CPU is in user mode, and 0 otherwise. Each U_k is Bernoulli with mean p_u. Thus, the Law of Large Numbers says that

$$p_u = \lim_{n\to\infty} \frac{U_1+U_2+\cdots+U_n}{n}$$

A good estimate of p_u then is

[1]This work was supported by the Director, Office of Energy Research, Scientific Computing Staff, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

$$p_u^* = \frac{U_1 + U_2 + \cdots + U_n}{n}$$

Taking the sample sequence to be Bernoulli assumes that the underlying process is *stationary*, which means that the probabilities of being in each state remain constant over time. Although this is not generally true of programs, there is no way to proceed without making this assumption. Programmers often put code to be profiled inside a loop, or otherwise run the code many times. This repetitive behavior then has an overall stationary behavior. Furthermore, long lived processes generally exhibit repetitive behavior, so the assumption is reasonable for CPU estimation as well.

The Conventional CPU Estimator

In the conventional method, rather than compute the probability estimates mentioned above, estimates of T_u and T_s are directly maintained. Call these times T_u^* and T_s^*. Assuming that the set of samples comprises a true random sample, this method would be equivalent to computing the probability estimates. Let Δ be the sampling interval. Then the algorithm computes T_u^* as

$$T_u^* = \sum_{k=1}^{n} \Delta U_k = (n\Delta)\left(\frac{1}{n}\sum_{k=1}^{n} U_k\right) = T_e^* p_u^* \quad (1)$$

since $T_e^* = n\Delta$ is an estimate of the elapsed time. Thus, T_u^* and $p_u^* T_e$ are approximately equal. A similar analysis holds for T_s^*.

This approach fails, however, because the samples used to compute T_u^* and T_s^* do not comprise a truly random sample. Since the sampling mechanism uses fixed intervals, random samples would only result if system and process behavior were itself random. The original implementation was probably based on this assumption. Unfortunately, programs are, for the most part, deterministic, so random behavior should not be expected. The BSD book [6] points out that the run time utilization estimates are in fact "statistical", but does not attempt to clarify the estimation technique.

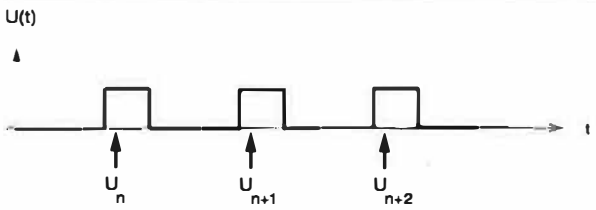


Figure 1: Process Isochrony

In any case, the statistical model allows us to clearly see where the conventional algorithm breaks down. The problem is that the sequence U_n appearing in Equation 1 is not a sequence of *independent* observations. For example, knowing the previous observation gives you information about the next one – i.e., adjacent observations are dependent. Since the Law of Large Numbers applies only to sequences of *independent* random variables, the probability estimate will not necessarily converge to the true probability.

The problem that arises from this lack of independence is clearly illustrated in the case of an isochronous process. A process is characterized as isochronous if its behavior is periodic and consistent, for instance, as shown by the graph in Figure 1. The function $U(t)$ represents the user mode utilization of the CPU, as a function of time, while the arrows indicated the fixed rate sampling process. Because the sampling process is synchronized with the process behavior, the estimator will compute $p_u^* = 1$, even though $p_u \neq 1$. This is a systematic error as it does not diminish with more samples.

Adding Randomization

Now that we know how the conventional estimator is failing, we would think it would be easy to correct the problem. All we need to do is change the sampling technique so that we get independent observations. This turns out to be a non-trivial problem.

In theory, the most straightforward approach would be to simply choose some number of random samples uniformly distributed over the lifetime of the system. But this approach is obviously infeasible since the system is continually running and its history is not retained.

An attractive alternative is to continue to use an interval based sampling approach, but to use random rather than fixed intervals. If we sample at a time T_i, then the next sample time is given by

$$T_{i+1} = T_i + W_{i+1}$$

where each W_i is a random variable, and $T_0 = 0$.

Intuitively, randomizing the sampling clock phase should break any synchronization with process behavior. But how can we be sure that the observations are statistically independent and hence that the sample mean will converge to the probability estimate? The answer depends on the distribution of W. For example, if W is constant, then we have the existing approach, which we know won't work. More generally, if W is *arithmetic*, for instance it takes on values only in $\{nk : n \geq 0, k = 0, 1, \ldots\}$, then we have a similar problem.

From a theoretical perspective, a particularly nice choice for W is the exponential distribution. The sequence T_k would then correspond to a Poisson process, for which a well known result is that the conditional distribution of arrivals on a subset of time, given their number, is uniform. In other words, if we know how many samples occurred over some interval, then those samples are uniformly distributed on that interval. These uniformly distributed random samples would result in a truly random aggregate sample.

However, implementation difficulties arise for exponentially distributed intervals. Occasionally, the time difference between adjacent samples will be smaller than the interrupt service time. Depending on how the clock hardware operates, race conditions could result when reprogramming the timer. Also, it is not clear what effect an occasional very large sample interval would have on other aspects of the system (for example, the scheduler).

Our solution is to let W be uniformly distributed on $[T_{min}, T_{max}]$. In this case, T_{min} can be chosen to be much larger than the interrupt service time, simplifying implementation.

We must be sure, however, that this approach, like the others, is unbiased. We can argue this using another result from probability theory, the Ergodic Theorem, which is a generalization of the Law of Large Numbers. This result says that the sample mean will converge to the true mean if the sequence of samples is ergodic (and not necessarily independent). Assuming the underlying process is stationary, and that our sampling process begins at $-\infty$, we can argue that our sample sequence is ergodic. We omit the details and refer the reader to [3, Ch. 6].

Note that the probability estimates converge independently of the frequency of the sampling clock. Only the *rate* of convergence is controlled by the mean sampling period. Thus, the average sampling rate, and hence the overhead of the CPU estimator, is dynamically adjustable. This contrasts with the existing system which required the rate to be configured into the kernel at compile time.

Implementation

Incorporating the randomized sampling model into the existing system was relatively straightforward. In the 4.4BSD kernel, all real-time and time-of-day events, including process scheduling, are driven off a fixed-rate *hardclock* interrupt. In the old system, the *hardclock* interrupt also gathered statistics; now they are driven off a separate *statclock* timer. Each time *statclock* returns from its interrupt context, the timer is reprogrammed for a random interval chosen from a uniform distribution as described in the previous section.

Process "wall clock time" is computed directly from the actual time of day at process switch. The *microtime* function is used to obtain a high resolution timestamp when the process is continued, and again when the process is suspended; the difference between these times is then added to a running sum. This figure becomes the T_e factor in the approximation formulas.

Meanwhile, on each *statclock* tick, the current process, if any, is charged a user, system, or interrupt tick according to the CPU mode at the time of the *statclock* interrupt; call these counts u, s, and i respectively. Since the sum of these counts is the total number of samples taken, the probability estimates are easily computed: $p_k = k/(u+s+i)$ for $k \in \{u, s, i\}$. From this and T_e, we can then compute T_u and T_s. T_i can be similarly computed, but since there is no way to identify the true source of such time it currently disappears into general system overhead.

The *statclock* abstraction is available only on machines with high-precision, programmable clocks. The randomized sampling intervals are generated by programming the clock's limit register with a pseudo-random number. To reduce overhead, a cheap-but-good random number generator is used [1]. On systems without programmable clocks, *statclock* is called directly from *hardclock*, and the functionality is unchanged from the existing system.

Code Profiling

Kernel support for user level code profiling [4] is carried out in a manner identical to CPU usage estimation. We therefore wanted to apply the lessons learned from the randomized sampling clock to the profiling system. Along the way, we fixed some problems with the traditional profiling support.

When a *statclock* tick occurs while a profiled process is running, a profiling buffer in user address space must be updated. In the previous system, this buffer update cannot be carried out by the clock interrupt handler because page faults are not permitted in an interrupt context. Instead, the clock handler schedules a profiling "asynchronous system trap" or AST, which causes a trap to occur just before returning to user mode. In this trap context, page faults may occur, and the user's profiling buffer is updated.

The 4.4BSD kernel avoids most such ASTs through two new routines to manipulate user memory from interrupt context. These routines attempt to update the user profiling counts directly. If a fault occurs, the update is aborted and the profiling code schedules an AST as before. Typically only a few such ASTs are required to page in the user profiling buffer. From then on, the updates can be carried out cheaply from interrupt context.

System call profiling in the existing system is inaccurate. In this case, rather than update the user profile buffer during the system call, the kernel reads the process's accumulated system time at entry to each call and again at exit from each call. The difference between these two times is converted to a tick count, which is added as if from an AST. This includes the same interrupt-time excess found in the per-process statistics, and computing this value is complicated due to the need to turn time into ticks.

In 4.4BSD, system call profiling is still done at the end of each call for efficiency, but now it is merely a matter of subtracting the previous system tick count from the current count.

Results

We devised three experiments, two contrived and one from production software, that uncover the anomalies of the old CPU utilization estimator. The tests were run under both SunOS 4.1.1 and 4.4BSD. In each experiment, the anomalies were clearly evident under the old CPU estimator; under 4.4BSD, they disappeared.

Interrupt Activity Interference

The first experiment clearly demonstrates that interrupt activity is charged to the current process as system time. A program was written that executes an infinite loop, and runs a 4 Hz alarm that logs system time usage. Since the program uses only a small amount of system CPU, just enough to process an alarm signal every 0.25 seconds, system time should accumulate *very* slowly.

This program was simultaneously run on two SPARCstation 1+ machines, one running SunOS 4.1.1 and the other 4.4BSD. Partway through execution, each host was exposed to an onslaught of interrupt activity[2]. The interrupt activity was then terminated, and finally, the program was stopped.

Figure 2 shows the plot of system CPU time versus real time for both processes. The lower line represents the 4.4BSD behavior, and is as expected – very little system time is accumulated. Under SunOS, however, during the interrupt activity, the process is charged with a significant amount of system CPU time. Note that this process had nothing to do with the interrupt activity; clearly, the statistics are skewed.

A CPU Adversary

An adversarial program was written in an attempt to defeat the CPU utilization statistics altogether. This program, which we call *hog*, first estimates the phase of the system clock. It then enters a hard loop, performing *gettimeofday* system calls, until just before a *hardclock* tick is going to happen. At this point, it goes to sleep until the next system clock. Thus, the process is never charged with a sampling tick, and never will accumulate CPU time.[3] Furthermore, its scheduling priority remains favorable, so it always runs, even if there are other processes waiting.

Since *hog* sleeps every other system clock tick, it will use at most half of the CPU. Thus, two *hogs* are required to use up the whole CPU. We augmented *hog* to fork once from *main*, and the results were dramatic. Table 1 shows timings for a CPU bound process when run in the presence and absence of *hog*. The CPU bound test program simply counted to 10 million. The first column gives the real time of execution, while the second column gives the CPU time as measured by the system. Without *hog*, the two systems are similar, as expected. But with *hog*, the SunOS process takes 80 times longer to finish even though the *time* command reports a utilization of 57%. If this figure were correct, it should only have taken 1.75 times longer. Under 4.4BSD, the test process gets a fair share of

[2]The interrupt activity was generated by putting each host's network interface into *promiscuous* mode, causing all network packets to be processed. A 780KB/s Ethernet transfer was then initiated on the local net.

[3]Actually, there is a low probability that the process is run just before a clock interrupt, in which case there is insufficient time to discover that the interrupt is coming.

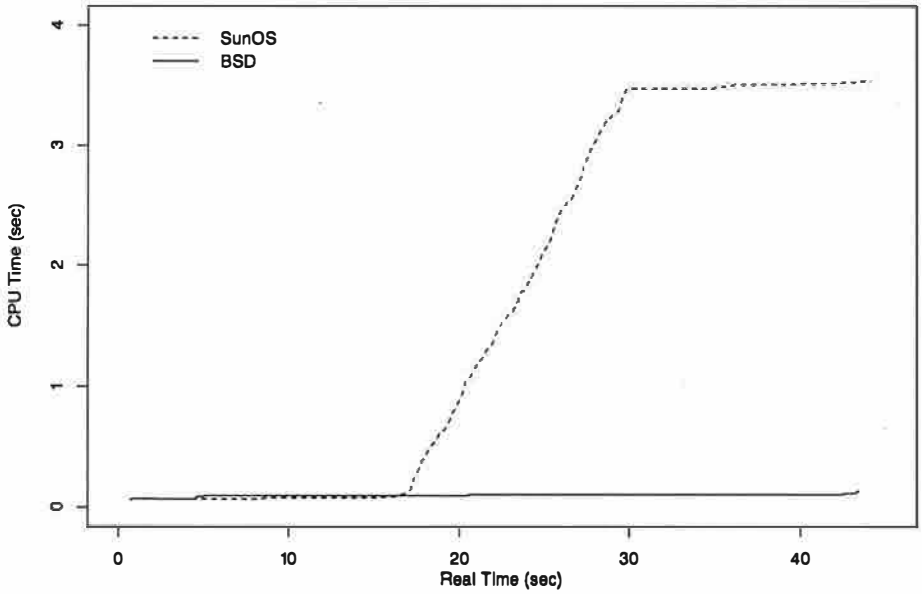


Figure 2: Interrupt Interference with System CPU Time

the CPU. The 15.5 seconds of real time is consistent with 45% utilization.

		real	*cpu*	*%cpu*
SunOS	w/o hog	7.1	7.1	100%
	w/ hog	584.5	334.8	57%
4.4BSD	w/o hog	7.1	7.0	99%
	w/ hog	15.5	7.0	45%

Table 1: Effect of hog on CPU bound process



Figure 3: Unexpected CPU Load Oscillations

In taking these measurements, we noticed anomalous scheduler behavior in 4.4BSD. Even though the CPU utilization estimates were accurate, the scheduler often exhibits unfairness between the CPU bound counting program and the *hog*. In the presence of the *hog*, the CPU bound process got as little as 9% and as much as 65% of the CPU. The BSD scheduler is known to be flawed [7], but it should do better here. This remains to be investigated.

The code for *hog* is given in the appendix.

The Isochronous Anomaly

The previous two experiments were run under controlled conditions in an attempt to expose the worst case behavior of the utilization error. However, we have experienced problematic behavior in a normal operating environment. For example, Figure 3 shows a window dump of *xcpu*, displaying a load oscillating between 0 and 10%, with a period of about one minute. The oscillations in the load average were not expected, and the cause was an audio conferencing program called *vat* [5, 2]. *Vat* processes a frame of audio samples every 22.5ms and should therefore exert a constant CPU load. But *xcpu* indicated otherwise.

To verify our theory that *vat* was causing these load fluctuations, we modified it to log its CPU time, every 22.5ms, to a debugging file, and ran the new version under both SunOS and 4.4BSD. Figure 4 shows these results. Since *vat* operates continuously, you would expect its CPU time to increase linearly. This is the case for 4.4BSD. But under SunOS, there are flat and steep regions of about 30 seconds in length. This anomaly is clearly due to the inaccuracy of the old sampling methodology.

The problem is that *vat* is running synchronously with the system clock. Since *vat* runs *exactly* every 22.5ms, it is aliased onto the 10ms system clock in only four possible slots.[4] As a result, when the minimum phase difference allows *vat* to carry out all of its processing before the clock ticks arrive, CPU time never accumulates. On the other hand, when the phase is such that ticks always occur, too much CPU time is charged. These two modes of operation are reflected in the SunOS data as the flat and steep regions. While this argument predicts that

[4]The least common multiple of the two periods is 90ms. Therefore, there are 90/22.5 = 4 phase positions for *vat* to cycle through.

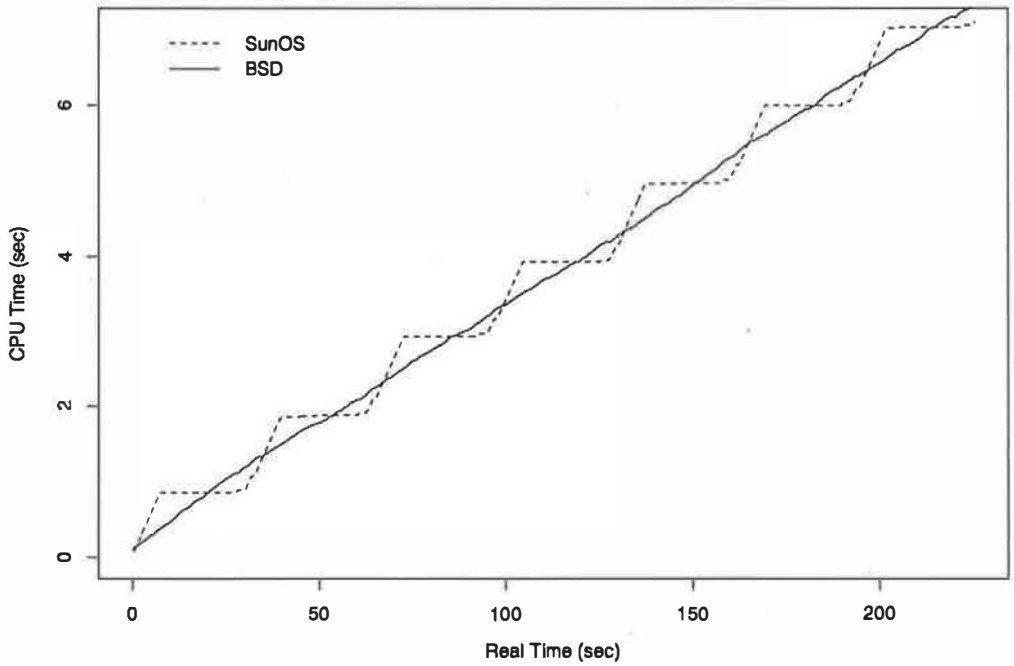


Figure 4: System Time Oscillations

we should remain in a given mode indefinitely, the data actually oscillates between the two modes with a period of about one minute. Without going into detail, various effects, some internal to *vat* and some due to unrelated interrupt activity, can cause the phase between *vat's* behavior and the system clock to drift.

Conclusion

We have presented a new approach for measuring CPU utilization that uses randomized sampling to overcome the deficiencies of the old approach. Randomization prevents an adversary from foiling the utilization estimator and precludes synchronization between the sampling system and process behavior. We have corrected problems with erroneous accounting of interrupt activity, and we have streamlined the kernel support for code profiling. Finally, we have conducted experiments to demonstrate that the new system performs as expected.

Acknowledgements

Van Jacobson originally suggested that randomization be used to circumvent the statistical biases in the CPU estimator. Additionally, he helped interpret the results of our experiments and provided suggestions for implementation strategies.

The idea that the CPU scheduler can be defeated is not new. Dheeraj Sanghi and Olafur Gudmundsson wrote a program similar to the hog presented in this paper. According to them, the idea was originally proposed by Ashok Agrawala.

We are grateful to Spyros Papadakis for helping us formulate our probabilistic arguments. His advice was impeccable; any errors were introduced by us.

Finally, we would like to thank Vern Paxson, Van Jacobson, Craig Leres, Deana Goldsmith, and the referees for their helpful comments on drafts of this paper.

Availability

The *statclock* code appears in the 4.4BSD alpha release. Currently, SPARCstation and HP-9000/300 series machines are supported.

Bibliography

[1] Carta, D. G. Two fast implementations of the "minimal standard" random number generator. *Communications of the ACM 33*, 1 (Jan. 1990).

[2] Casner, S., and Deering, S. First IETF Internet audiocast. *ConneXions 6*, 6 (1992), 10-17.

[3] Durrett, R. *Probability: Theory and Examples.* Brooks/Cole Publishing Company, 1991.

[4] Graham, S. L., Kessler, P. B., and McKusick, M. K. gprof: A call graph execution profiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction* (June 1982).

[5] Jacobson, V., and McCanne, S. *The Vat Manual Page.* Lawrence Berkeley Laboratory, Berkeley, CA, Feb. 1992. Available via anonymous ftp to ftp.ee.lbl.gov.

[6] Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, 1989.

[7] Straathof, J. H., Thareja, A., and Agrawala, A. UNIX scheduling for large systems. In *Proceedings of the 1986 Winter USENIX Technical Conference* (Denver, CO, Jan. 1986), USENIX, pp. 111-138.

Author Information

Steven McCanne has been with the Lawrence Berkeley Laboratory since 1988, working on network analysis tools and remote conferencing applications. He holds a B.S. degree in Electrical Engineering and Computer Science from U.C. Berkeley, and is currently a Ph.D. student in Computer Science at U.C.B. His email address is mccanne@ee.lbl.gov.

Chris Torek has been rewriting bits of the Berkeley kernel for about six years. He joined LBL in 1991, and has been working on porting 4.4BSD to the SPARCstation. In his off hours he spends far too much time on USENET. Reach him electronically at torek@ee.lbl.gov.

Reach both authors at:

Lawrence Berkeley Laboratory
One Cyclotron Road
Berkeley, CA 94720

Appendix: Adversary Source Code

```
#include <signal.h>
#include <sys/param.h>
#include <sys/time.h>
#include <sys/resource.h>

#define tvdiff(x, y) \
    (1000000 * ((y).tv_sec - (x).tv_sec) + (y).tv_usec - (x).tv_usec)

struct timeval hc;      /* our best guess for when a hardclock happened */
struct timeval now;     /* hold time-of-day for signal handler */
volatile int ntick;

alarm_handler()
{
    u_long u;
    struct timeval tv;
    static int mindel = 5000000;

    ++ntick;
    gettimeofday(&tv, 0);
    u = tvdiff(now, tv);
    if (u < mindel) {
        mindel = u;
        hc = tv;
    }
    usleep(1);
}

/*
 * Try to figure out when hardclock happens.
 */
struct timeval
train()
{
    struct itimerval it;

    signal(SIGVTALRM, alarm_handler);
    it.it_interval.tv_usec = it.it_value.tv_usec = 1;
    it.it_interval.tv_sec = it.it_value.tv_sec = 0;
    /*
     * Sleep right before we set the timer. This way, we're sure to
     * get a whole time slice, and we won't be switched out before
     * we estimate the hardclock time.
     */
    usleep(1);
    setitimer(ITIMER_VIRTUAL, &it, 0);
    for (ntick = 0; ntick < 20; )
        gettimeofday(&now, 0);
    /*
     * Turn the timer off.
     */
    it.it_interval.tv_usec = it.it_value.tv_usec = 0;
    setitimer(ITIMER_VIRTUAL, &it, 0);

    return (hc);
}
```

```
int
main(argc, argv)
    int argc;
    char **argv;
{
    long us, s, bias, delta, off;
    struct timeval tv;

    /*
     * Determine when hardclocks are happening then compute a bias
     * with respect to an even multiple of hardclock ticks.
     * Assume 10ms tick.  Since a second is an even multiple of
     * a tick, we only need to look at usecs.
     */
    tv = train();
    bias = tv.tv_usec % 10000;

    /*
     * Make one copy of ourself.
     * We need two processes to do real damage.
     */
    fork();

    for (;;) {
        gettimeofday(&tv, 0);
        /*
         * Round down to even tick multiple, then add in bias.
         * Compute estimate of next hardclock into s and us.
         */
        s = tv.tv_sec;
        us = tv.tv_usec;
        delta = us % 10000;
        off = bias - delta;
        if (off < 0)
            off += 10000;
        us += off;
        if (us >= 1000000) {
            us -= 1000000;
            ++s;
        }
        /*
         * Spin until 1ms before next hardclock.
         */
        us -= 1000;
        while (tv.tv_sec < s || (tv.tv_sec == s && tv.tv_usec < us))
            gettimeofday(&tv, 0);
        usleep(1);
    }
}
```

Fault Interpretation: Fine-Grain Monitoring of Page Accesses

Daniel R. Edelson – INRIA Project SOR

ABSTRACT

This paper[1] presents a technique for obtaining fine-grain information about page accesses from standard virtual memory hardware and Unix operating system software. This can be used to monitor all userode accesses to specified regions of the address space of a process. Application code can intervene before and/or after an access occurs, permitting a wide variety of semantics to be associated with memory pages.

The technique facilitates implementing complex replication or consistency protocols on transparent distributed shared memory and persistent memory. The technique can also improve the efficiency of certain generational and incremental garbage collection algorithms. This paper presents our implementation and suggest several others. Efficiency measurements show faults to be about three orders of magnitude more expensive than normal memory accesses, but two orders of magnitude less expensive than page faults. Information about how to obtain the code via anonymous ftp appears at the end of the paper.

Introduction

This paper shows how a program can use common Unix virtual memory page protection and signal handling to monitor all accesses to selected pages of its address space. The technique has been encapsulated in a library called FI for Fault Interpretation. We discuss a number of applications for this technique including garbage collection and consistency/replication protocols for transparent distributed shared memory.

Virtual memory page protection has been used for similar reasons before [AEL88, AL91, DWH+90]. The difference with our approach is that most other techniques unprotect a protected page when a fault occurs. For some period of time thereafter, there is no monitoring of how many times and at what addresses the page is accessed. With fault interpretation, in contrast, a page does not remain unprotected. When an access causes a fault, the page is now unprotected and the access is performed. Then, the page is restored to its previous protection state and the application resumes at the subsequent machine instruction. A notification function, registered by the application, can intervene immediately before and/or after the access. It is as if the access succeeds and the application is informed that the access occurs.[2]

[1]This work was performed while the author was visiting INRIA. The author's most recent affiliation is: Computer and Information Science, University of California, Santa Cruz CA 95064, daniel@cse.ucsc.edu.

[2]Caveat: This technique requires knowing the precise state of the CPU when a protection violation occurs. It may not be possible to implement this functionality on all RISC architectures. We have implemented it on the SPARC processor [Cyp90, Sun87].

The remainder of this report is organized as follows: Section 1 gives an overview of the technique with a small example. Section 2 presents a C library interface that encapsulates the functionality. Section 3 discusses some applications. Then, Sect. 4 describes the implementation and Sect. 5 presents efficiency measurements. Section 6 discusses the availability of the library and some caveats, and Sect. 7 concludes the report.

1 Fault Interpretation: Memory Access Monitoring

Fault interpretation allows an application to detect all reads and/or writes to selected pages of its virtual address space. The library uses the mprotect system call to disallow accesses to monitored pages. An access to a protected page causes a fault, which Unix passes to the application as a signal. The FI signal handler unprotects the page and notifies the application of the access. Then, the faulting instruction is restarted; it succeeds because the page is unprotected. Control returns immediately to the FI library, which notifies the application again, reprotects the page, and resumes the application at the next instruction.

As above, the application can be notified twice per access. These two function invocations are referred to as pre-access notification and post-access notification. The two calls permit a wide variety of semantics, for example, pre-access notification might be used to read a page over the network, to obtain a write-lock on a page, or simply to record the address of the access. Post-access notification might release a write lock or send an updated page to other hosts. It might also be used by a debugger to detect that a variable has been accessed.

Arguments to the notify function indicate the address of the access, its type (read, write or swap) and how many bytes are involved; the access type and number of bytes are obtained by decoding the instruction. During notification, the accessed page is unprotected, permitting the notify function to access the page without faulting.

FI utilizes the Unix mprotect system call and traps the resulting signal, which is typically either SEGV or BUS. When the signal is caught, the operating system passes the handler information about the faulting context. To support fault interpretation, this information must include the program counter and the other registers. FI uses this information to determine what access the program was performing and to alter the usual flow of control.

The FI signal handler can coexist peacefully with other signal handlers, provided they are not both trying to catch the same kinds of signals on the same memory pages. When FI traps a signal from a fault on an unmonitored page, the signal is propagated to any other handler that is installed for the signal.

The biggest difference between this and common uses of virtual memory (VM) protection is that the faulting instruction is (effectively) single-stepped, rather than resumed normally. After the instruction succeeds, control returns to the library's reprotect block, which performs the post-access notification, reprotects the page, and resumes the application. Thus, the page is only unprotected for the one instruction that faults (as well as during notification); all accesses to the page can be trapped. This effect could be accomplished using the ptrace system call but that doesn't permit a process to monitor itself; it can only monitor another process.

The best way to demonstrate the exact effect obtained is through an example. We present a small

```
#include <stdio.h>
#include "fi.h"
#define PGSIZE 4096

/* notify prints the address and type of the access */
void notify(void * addr, size_t nb, fi_flags_t type) {
   printf("NOTIFY: Access 0x%p for %d bytes, type ",addr,nb);
   if (type & FI_PREREAD)    printf("PREREAD ");
   if (type & FI_PREWRITE)  printf("PREWRITE ");
   if (type & FI_POSTREAD)  printf("POSTREAD ");
   if (type & FI_POSTWRITE) printf("POSTWRITE ");
   printf(""n");
}

int main()  {
    int i, * addr;

    fi_initialize();
    /* Allocate one page of managed memory */
    addr = (int*) fi_alloc(PGSIZE,fi_noaccess,notify,FI_ALL);
    printf("Causing four faults now!"n");
    addr[0] = 6;
    addr[121] = 999;
    i = addr[40];
    i = addr[400];
    printf("Permit READ accesses without faulting."n");
    fi_setprot(addr, PGSIZE, fi_readonly);
    printf("Causing two faults now!"n");
    addr[0] = 6;
    addr[121] = 999;
    i = addr[40];     /* no fault: read access permitted */
    i = addr[400];    /* no fault: read access premitted */
    fi_free(addr);
    return 0;
}
```

Figure 1: A small FI application

test application that obtains some protected memory and causes faults. The handler displays the address and the type of every fault. The application is shown in Fig. 1. The output of the application follows in Fig. 2. As this example demonstrates, an application can very easily obtain a region of managed memory. Thereafter, the application will be notified upon every access to the region.

2 Library

The FI library encapsulates the functionality that is described in the previous section. The library includes calls to obtain managed memory, to change the state or attributes of the memory, and to release the memory when it is no longer needed.

When a fault occurs, the exact sequence of events is the following:

1. An instruction attempts to access a protected page; the instruction faults. The operating system invokes the FI-installed signal handler.
2. The FI signal handler verifies that the fault occurred on a page that is managed by FI. If not, the signal is propagated to any previously installed signal handler. If the page is managed, the page will not be unprotected.
3. If pre-access notification has been requested, the application is notified. The notification function is passed the fault address, the number of bytes, and flags indicating whether the access is a read or write (or both) and that the notification is pre-access. The notification function can examine or modify the page.
4. The faulting instruction is executed again. Since the page is not protected, the access succeeds. Control returns immediately to FI.
5. If post-access notification has been requested, FI calls the notification function. The same arguments are passed except that the flags indicate post-access.
6. FI returns the page to its previous protection state and resumes the application. The application continues with the instruction following the one that caused the fault.

The library is written in C [ANS89, ISO90] using Unix system call extensions. It can also be compiled as C++ code. In order to avoid name clashes, all external identifiers used in FI begin with "fi_".

Managed memory is obtained in segments whose size is an integral number of pages. Within a segment, the protection state, notification, and notify function of each page may be independently specified.

The library interface defines a small number of types, constants and functions. The first type is an enumeration that indicates what protection state the application requires for a page. The type is defined as follows:

```
typedef enum {
    fi_noaccess,
    fi_readonly,
    fi_readwrite
} fi_prot_t;
```

The enumeration constants mean:

fi_noaccess No accesses to the page are permitted, meaning all accesses result in faults.

fi_readonly Read accesses do not fault.

fi_readwrite Both reads and writes are permitted without faulting.

Another set of flags defines the types of notification. The flags are bit values that may be ORed together. The values of the constants have been elided; see Figure 4.

When obtaining pages of managed memory, the application supplies a pointer to a notification function. The type of that function pointer is the following:

```
Causing four faults now!
NOTIFY: Access at 0x7000 for 4 bytes of type PREWRITE
NOTIFY: Access at 0x7000 for 4 bytes of type POSTWRITE
NOTIFY: Access at 0x71e4 for 4 bytes of type PREWRITE
NOTIFY: Access at 0x71e4 for 4 bytes of type POSTWRITE
NOTIFY: Access at 0x70a0 for 4 bytes of type PREREAD
NOTIFY: Access at 0x70a0 for 4 bytes of type POSTREAD
NOTIFY: Access at 0x7640 for 4 bytes of type PREREAD
NOTIFY: Access at 0x7640 for 4 bytes of type POSTREAD
Change page to READONLY.
Causing two faults now!
NOTIFY: Access at 0x7000 for 4 bytes of type PREWRITE
NOTIFY: Access at 0x7000 for 4 bytes of type POSTWRITE
NOTIFY: Access at 0x71e4 for 4 bytes of type PREWRITE
NOTIFY: Access at 0x71e4 for 4 bytes of type POSTWRITE
```

Figure 2: Output of the small FI application

```
typedef void (*fi_notify_t)(caddr_t,
                  size_t, fi_flags_t);
```

The caddr_t argument is the address of the fault. The size_t argument is the number of bytes involved in the access. The fi_flags_t argument indicates the type of access and whether the notification is pre-access or post-access.

Finally, the last part of the interface is the prototypes of the library functions. These prototypes are summarized in Fig. 3. The meanings of the functions are the following:

fi_initialize This function must be called first to initialize the library.

fi_alloc This routine allocates new monitored memory. The function returns a pointer to the allocated pages. The initial protection state and notify function are parameters to the function, as is the number of pages to allocate.

fi_addpages As with fi_alloc this function adds more managed memory. However, this routine allows the user to supply the address of the memory, rather than obtaining the memory from valloc or sbrk.

fi_free This free routine tells the library to stop using a set of pages. If the pages were obtained with fi_alloc they are deallocated.

fi_setprot This function sets the protection state of one or more managed pages. This determines what kinds of accesses, reads or writes, cause faults.

fi_setnotify This function sets the notify function pointer associated with one or more pages.

fi_setflags The fi_setflags interface is used to set the kind of notification required: pre-access and/or post-access.

fi_getprot This function returns the protection state of a page.

fi_getnotify This function returns the notify function pointer associated with a page.

fi_getflags This routine returns the notification flags of a page.

3 Applications

Possible applications of this technique include: write-detection in generational or incremental garbage collection, and consistency/replication protocols for shared memory.

Generational Garbage Collection

The idea behind generational garbage collection (GC) is that some objects are likely to remain reachable for the immediate future, thus, attempting to reclaim their memory is not worthwhile. [DWH+ 90, LH83, Moo84]. Typically, young objects are expected to become garbage relatively soon [Ung84], therefore, the garbage collector concentrates its effort on the young objects.

A garbage collection of the young objects (the younger generation) requires locating all pointers to young objects. Such pointers are of three types:

```
void  fi_initialize(void);
void* fi_alloc(size_t, fi_prot_t, fi_notify_t, fi_flags_t);
void* fi_addpages(void*, size_t, fi_prot_t, fi_notify_t, fi_flags_t);
int   fi_free(void* addr);
int   fi_setprot(void* pgaddr, size_t nb, fi_prot_t nw);
int   fi_setnotify(void* pageaddr, size_t nb, fi_notify_t nw);
int   fi_setflags(void* pgaddr, size_t nb, fi_flags_t nw);
int   fi_getprot(void* pgaddr, fi_prot_t* old);
int   fi_getnotify(void* pageaddr, fi_notify_t* old);
int   fi_getflags(void* pgaddr, fi_flags_t* old);
```

Figure 3: FI function prototypes

```
typedef unsigned char fi_flags_t;
#define FI_PREREAD   /* Pre-access notification for reads         */
#define FI_PREWRITE  /* Pre-access notification for writes        */
#define FI_PRE       /* Pre-access notification for all accesses  */
#define FI_POSTREAD  /* Post-access notification for reads        */
#define FI_POSTWRITE /* Post-access notification for writes       */
#define FI_POST      /* Post-access notification for all accesses */
#define FI_READ      /* Pre and post notification for reads       */
#define FI_WRITE     /* Pre and post notification for writes      */
#define FI_ALL       /* Pre and post notification for all accesses*/
```

Figure 4: Types of notification

1. pointers on the stack, in global data, and in registers,
2. pointers in young objects, or,
3. pointers in old objects.

Pointers of the first two types are common to all GC algorithms and do not introduce new difficulties. Pointers of the third kind are called back pointers and they introduce a problem that is unique to generational garbage collectors. These pointers must be located to avoid erroneously reclaiming live objects. However, since the collector is concentrating on young objects, it does not want to examine the old objects to locate these pointers. Thus, the task is to efficiently locate the set of all these pointers.

Some collectors add a run-time test to every (pointer) assignment to see if a back pointer is being created. Other collectors do not attempt to locate each individual pointer, but rather identify the set of pages that might contain such pointers, the remembered pages. During garbage collection, every object on a remembered page is scanned for back pointers. This has been implemented using page protection [DWH+ 90]. The garbage collector write-protects all of the older-generation pages. Every fault indicates that there has been an assignment to an older generation object; the page is added to the remembered set. Upon collection, the remembered pages are scanned for back pointers. If a page contains no back pointers, then then it is deleted from the remembered set. Otherwise, it is left in the set.

This implementation of the remembered set unprotects a page every time a fault occurs, permitting any number of writes to the page. Since it doesn't know what addresses were written, the collector must scan every object on every remembered page looking for back pointers. Even if only one word on the page is modified, the collector still must check every field of every object. In contrast, through memory access monitoring, the collector can have available the exact list of address that are modified. It is not necessarily desirable to remember the exact list, since that could be quite expensive. Instead, the collector can keep N remembered addresses per page. For the first N faults that occur on a page, the collector stores the fault address. Upon the next fault after that, the collector unprotects the page and treats the page the same as in the old system. This bounds the maximum time and space overhead due to faulting.

The exact value of N depends on two things: the efficiency of handling a fault, and the cost of scanning a page. If every field on a page can be scanned in less time than it takes to handle a fault, then fault interpretation should not be used. However, if scanning objects is relatively expensive, then remembering several stored addresses may improve efficiency.

Incremental Garbage Collection

Incremental garbage collection is a family of algorithms in which the collector never stops the application for an extended period of time. The first such algorithm was Baker's copying collector [Bak78] with many other algorithms based on it. To avoid annoying pauses, the collector does its work in short chunks. Incremental garbage collectors are often concurrent, in which case protected pages of memory can serve as medium grain synchronization mechanism between the collector and the application [AEL88].

Incremental Mark-and-Sweep Collection

Incremental mark-and-sweep garbage collection has been implemented previously using virtual memory page protection [BDS91]. The normal implementation provides one bit of information per page: there was or was not a fault. Pages on which a fault occurred must be entirely rescanned. This is another case in which fault interpretation can provide finer granularity information, possibly increasing the efficiency of the algorithm.

Incremental mark-and-sweep collectors do their work in short bursts. During each burst, the collector follows pointers and may discover that some additional objects are accessible. The collector marks the accessible objects so they will not be deallocated at the end of the collection. After chasing some pointers and marking some objects, the cycle ends and the collector returns control to the application. A burst in which the collector runs out of pointers signals the end of the mark phase.

Each time the collector returns control to the application, the application is free to modify marked objects. The application may store in a marked object the only pointer to an unmarked object. If the collector never again examines the marked object, the pointer won't be discovered: the unmarked object remains unmarked and is incorrectly deallocated by the collector. Thus, marked objects that are subsequently modified must be reexamined.

VM protection can be used to detect this case. Any page that contains marked objects is write-protected. If a fault occurs, the page is flagged. After the mark phase has nominally finished, all the flagged pages are scanned for marked objects with pointers to unmarked objects. When any such pointer is found, the data structure reachable from the pointer is marked.

Fault interpretation can be used to remember the first N fault addresses per page. Only N addresses per page are remembered to bound the total time spent servicing faults. After the mark phase has terminated, the pages that had between 1 and N faults can be serviced very quickly because the addresses of the writes have been saved.

Consistency and Replication Control

FI can be used to implement arbitrary replication and consistency protocols on top of transparent distributed shared memory [LH89]. The contribution of FI is the ability to execute application code before and after memory pages are accessed. This code might, for example, implement a voting algorithm [Lon88]. The consistency protocol runs transparently; the client accesses the memory with normal load and store instructions.

One possible implementation is the following. Shared memory pages are replicated on all the participating sites. Upon a write, the pre-access handler of the process that is writing sends packets over the network to lock the location. When the lock is obtained, the write executes. Then, the post-access handler unlocks the location. For reads, if there are currently no locks on a page, the page does not need to be read-protected. If there is at least one lock on a page, the page is protected so that read accesses can't occur concurrently with a write access at the same location. The pre-access handler for reads checks that the location is not locked, and if it is not, allows the read to complete. If the location is locked, the handler blocks until the location is unlocked. Post-access read notification is not required.

4 Implementation

There are a number of ways that fault interpretation can be implemented. By and large, they are architecture specific and require reading the state of the CPU when the fault occurs. Thus, this technique is less portable and less general than those discussed by Appel [AL91]. Nonetheless, it has several uses and may let some programs run more efficiently.

Code Modification

When the signal handler is invoked after a fault, it determines what instruction has faulted. The instruction immediately following the faulting instruction is overwritten with an unconditional branch to the block of handler code called the reprotect block.2 Then, the signal handler unprotects the page and returns, allowing the operating system to resume the application.

When it resumes, the application re-executes the instruction that caused the fault. Since the page is now unprotected, this succeeds. Then, control follows the branch to the reprotect block. This block performs post-access notification, reprotects the page, restores the instruction sequence that was modified, and branches back to the application.

Register Modification

The SPARC architecture permits a much simpler implementation that does not require code modification. The SPARC has a register called npc for next program counter. This register contains the address of the instruction that will execute after the current instruction completes. This register is used to implement delayed branches[3]. The npc register makes it particularly easy to implement FI.

Upon a fault, the signal handler can read and modify the CPU state at the faulting instruction. This state includes the contents of npc. The previous value of this register is saved, and the address of the reprotect block is assigned to the register. Then, the signal handler unprotects the page and returns. The application again executes the instruction that faulted; this time the access succeeds. Since npc points to handler code, control jumps to the reprotect block. As before, the application is notified, the page is restored to its former protection state, and control branches back to the

Instruction interpretation

Another way of executing a single instruction is to parse and interpret the instruction. On a RISC processor this is not very difficult or inefficient, provided the operating system makes the entire context of the faulting instruction available. This also requires being able to restart the instruction following the faulted instruction. One advantage of this is the interpreter can take advantage of extra information. For example, if the fault page is also mapped without protection elsewhere in the address space [AL91, Wil92], the interpreter can use that version to avoid needing to unprotect and reprotect the page.

Parallelization

The FI code is currently sequential. However, the majority of it could be parallelized. There are two main issues that must be resolved. The first is the use of global data. Two parts of the FI library communicate through global variables. In a parallel implementation, this data would have to be replicated on a per-thread basis.

The second issue is the following: If any thread is executing when a page is unprotected, the thread can access the page without being monitored. Thus, whenever FI unprotects a page, it must first stop all the threads in the system. They remain stopped until the page's protection is restored.

5 Efficiency

The key operations in terms of efficiency are changing the protection state of a page and handling a fault. The times for these two operations are presented in Table 1. The timing information was obtained with the SunOS version 4.1.1 getrusage system call. The tests were performed on a Sun IPX with a cycle time of 25 ns (40Mhz). The cycles-per-operation figures are obtained by dividing the time per operation by the cycle time.

[3]On delayed branch architectures, a nop is written after the branch. application. This is the implementation used in the current version of the FI library.

The time to protect a page was obtained by making the mprotect system call in a loop. The time this call requires to execute depends on whether the page in question is accessed or not, and whether it is clean or dirty. Therefore, this test was repeated for unaccessed pages, pages that had been read from, and pages that had been written to. In each case, the page was entirely initialized to zeros before beginning the test. The data for each class of page are presented. This was repeated several times with the total time and the total number of iterations summed and averaged.

The time for handling a fault was obtained by writing a fault handler that leaves the page protected N - 1 times so that restarting the instruction causes another fault. Then, on the Nth iteration, the handler unprotects the page and the instruction completes successfully.

The time for protect+fault+unprotect was obtained by protecting a page, faulting, and unprotecting the page, all in a loop. This is a test whose efficiency is also measured in [AL91] and is repeated here to provide a baseline for comparison.

The time for fault interpret is the time to interpret a fault, i.e., to access a protected page and have the application's notify function informed that the access has occurred, while finishing with the page still protected. This consists of fault+unprotect+protect+small overhead. The application's notify function for this test returns immediately.

Lastly, we present the time for handling a page fault. This data was obtained obtained by allocating more virtual memory than the machine has physical memory and repeatedly sequentially touching every page. This was done once with pages being read and once with pages being read and written. In both cases, page accesses are sequential. This information is provided to offer a comparison between the efficiency of handling protection faults and page faults.

The data show that this implementation of fault interpretation is about 5% more expensive than standard fault handling (for substantially greater functionality). Nonetheless, protection faults are very expensive, costing approximately 20,000 cycles each. This cost in terms of memory references is much different, of course, probably closer to 8000 memory references. Therefore, if taking a fault can save more than 8000 memory references, there will be an increase in efficiency.

What is really clear is how expensive page faults are. If we can save a single page fault, then we can interpret over 30 protection faults and still see an increase in efficiency (based on the relative costs of a page fault and fault interpretation). A generational collector that stores generation counters in objects, or an incremental mark-and-sweep collector that stores mark bits with objects, could significantly improve efficiency with fault interpretation. For transparent persistent memory, the fault time is inconsequential compared to the time to write the data to disk. Similarly, assuming that the time for a network message for a relatively fast protocol such as UDP is on the order of 1.5ms [Mak89], fault handling should not be the bottleneck in implementing distributed shared memory.

Lastly, we observe that disk and network latencies do not scale with processor speeds, whereas fault handling latency does increase with faster CPUs, subject to memory access time. Thus, relative to disk and network I/O, the efficiency of fault interpretation will improve with faster CPUs. It will also improve if operating system implementors provide faster trap handling.

Operation	Count	Total Time	Time per Operation	Cycles per Operation
mprotect, unaccessed	80,000	4.0s	50s	2000
mprotect RO-RW, clean	80,000	14.4s	179s	7160
mprotect RW-RW, clean	80,000	22.1s	275s	11000
mprotect RW-RW, dirty	80,000	21.2s	265s	10600
handle a fault	500,000	81.7s	163s	6520
protect+fault+unprotect	500,000	258.5s	517s	20680
fault interpret	500,000	270.0s	540s	21600
page fault, reading	20,480	480.0s	23,437s	937,480
page fault, writing	20,480	757.0s	36,963s	1,478,520

Table 1: Efficiency of the component operations. The measurements were taken on a 40Mhz Sun IPX. Unaccessed means the page has neither been read nor written. Clean means the page has been read since the last call to mprotect. Dirty means the page has been written since the last call to mprotect. RW-RW means successive calls to mprotect always grant full access to the page. RO-RW means successive calls to mprotect alternate between restricting access and restoring access.

6 Availability

The FI library has been implemented for the SPARC processor. The code will compile either as an ANSI/ISO C program or as a C++ program. The source code is available via anonymous ftp from ftp.cse.ucsc.edu (128.114.134.19) in pub/csl/vm-trace.tar.Z. It can also be obtained from ftp.inria.fr (128.93.1.26) in INRIA/c++-gc/vm-fault.tar.Z. The code is not public domain, but may be used without fee for any purpose, commercial or otherwise.

All of the test programs that were used for our efficiency measurements are available with the library. The names of the files (and their purposes) are as shown in Table 2.

File	Purpose
t0.c	Measure the efficiency of mprotect
t1.c	Sample FI application, obtain and exercise managed memory
t2.c	Measure the efficiency of trapping the signal upon a memory protection fault
t3.c	Measure the time required to protect a page, fault on it, and then unprotect it
t4.c	Measure the time required to interpret a fault
t5.c	Measure the time required to handle a page fault when reading sequential pages
t6.c	Measure the time required to handle a page fault when writing sequential pages
t7.c	Measure the efficiency of mprotect (more detail than t0.c)

Table 2: Files and purposes

7 Conclusion

We present a library that provides more functionality than is usually obtained from standard virtual memory hardware and operating system software. Given sufficiently fast trap handling, this technique can be used to improve the efficiency of incremental or generational garbage collectors. It may also be useful for persistent object stores, coherent distributed shared virtual memory, and other algorithms.

Acknowledgements

This work was supported in part by Esprit project 5279 Harness.

References

[AEL88] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. In Proc. PLDI '88, pages 11-20, July 1988. SIGPLAN Not. 23(7).

[AL91] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In ASPLOS Inter. Conf. Architectural Support for Programming Languages and Operating Systems, pages 96-107, Santa Clara, CA, April 1991. SIGPLAN Not. 26(4).

[ANS89] ANSI X3.159-1989, 1989. American national standard for the C programming language.

[Bak78] H. G. Baker. List processing in real time on a serial computer. Communications of the ACM, 21(4):280-294, April 1978.

[BDS91] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In Proc. PLDI '91, pages 157-164. ACM, June 1991. SIGPLAN Not. 26(6).

[Cyp90] Cypress Semiconductor. SPARC risc users guide, 1990.

[DWH+ 90] Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: Framework and implementations. In Proc. POPL '90, pages 261-269. ACM, January 1990.

[ISO90] ISO 9899-1990, 1990. International standard for the C programming language.

[LH83] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. Communications of the ACM, 26(6):419-429, June 1983.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. ACM Transactions on Computer Systems, 7(4):321-359, November 1989.

[Lon88] Darrell D. E. Long. The Management of Replication in a Distributed System. Ph.D. dissertation, University of California at San Diego, August 1988.

[Mak89] Mesaac Mounchili Makpangou. Protocoles de communication et programmation par objets : l'exemple de SOS. PhD thesis, Universite Paris VI, Paris (France), February 1989.

[Moo84] David Moon. Garbage collection in a large LISP system. In Symp. Lisp and Functional Programming, pages 235-246. ACM, 1984.

[Sun87] Sun Microsystems, Inc. The SPARC architecture manual, 1987. Part No. 800-11399-07.

[Ung84] David Ungar. Generation Scavenging: A non-disruptive high performance storage reclamation algorithm. In ACM SIGPLAN/SIGSOFT Symp. Practical Software Development Environments, pages 157-167. ACM, April 1984. SIGPLAN Not. 19(2).

[Wil92] Paul Wilson, 1992. Personal communication.

Author Information

The author is a Ph.D. student in Computer and Information Science at the University of California, Santa Cruz. He plans to graduate in 1993. From 1991 to 1992, he spent 12 months as a visiting researcher at INRIA Rocquencourt, where this work

was done. He also programs C++ on contracts and teaches C++ classes in industry. Formerly, he was an engineer in the operating systems group at the Santa Cruz Operation. Reach him electronically at Daniel.Edelson@inria.fr. Reach him via paper mail at:

INRIA Project SOR
Rocquencourt B.P. 105
78153 Le Chesnay CEDEX
FRANCE

UNIX disk access patterns

Chris Ruemmler and John Wilkes – Hewlett-Packard Laboratories

ABSTRACT

Disk access patterns are becoming ever more important to understand as the gap between processor and disk performance increases. The study presented here is a detailed characterization of every low-level disk access generated by three quite different systems over a two month period. The contributions of this paper are the detailed information we provide about the disk accesses on these systems (many of our results are significantly different from those reported in the literature, which provide summary data only for file-level access on small-memory systems); and the analysis of a set of optimizations that could be applied at the disk level to improve performance.

Our traces show that the majority of all operations are writes; disk accesses are rarely sequential; 25–50% of all accesses are asynchronous; only 13–41% of accesses are to user data (the rest result from swapping, metadata, and program execution); and I/O activity is very bursty: mean request queue lengths seen by an incoming request range from 1.7 to 8.9 (1.2–1.9 for reads, 2.0–14.8 for writes), while we saw 95th percentile queue lengths as large as 89 entries, and maxima of over 1000.

Using a simulator to analyze the effect of write caching at the disk level, we found that using a small non-volatile cache at each disk allowed writes to be serviced considerably faster than with a regular disk. In particular, short bursts of writes go much faster – and such bursts are common: writes rarely come singly. Adding even 8KB of non-volatile memory per disk could reduce disk traffic by 10–18%, and 90% of metadata write traffic can be absorbed with as little as 0.2MB per disk of non-volatile RAM. Even 128KB of NVRAM cache in each disk can improve write performance by as much as a factor of three. FCFS scheduling for the cached writes gave better performance than a more advanced technique at small cache sizes.

Our results provide quantitative input to people investigating improved file system designs (such as log-based ones), as well as to I/O subsystem and disk controller designers.

Introduction

The I/O gap between processor speed and dynamic disk performance has been growing as VLSI performance (improving at 40–60% per year) outstrips the rate at which disk access times improve (about 7% per year). Unless something is done, new processor technologies will not be able to deliver their full promise. Fixes to this problem have concentrated on ever-larger file buffer caches, and on speeding up disk I/Os through the use of more sophisticated access strategies. Surprisingly, however, there has been very little published on the detailed low-level behavior of disk I/Os in modern systems, which such techniques are attempting to improve. This paper fills part of this void, and also uses the data to provide analyses of some techniques for improving write performance through the use of disk caches.

We captured every disk I/O made by three different HP-UX systems during a four-month period (April 18, 1992 through August 31, 1992). We present here analyses of 63-day contiguous subsets of this data. The systems we traced are described in Table 1: two of them were at HP Laboratories, one (snake) at UC Berkeley.

The most significant results of our analyses of these systems are: the majority of disk accesses (57%) are writes; only 8–12% of write accesses, but 18–33% of reads, are logically sequential at the disk level; 50–75% of all I/Os are synchronous; the majority (67–78%) of writes are to metadata; user-data I/Os represent only 13–41% of the total accesses; 10–18% of all write requests are overwrites of the last block written out; and swap traffic is mostly reads (70–90%).

Name	*Processor*[a]	*MIPS*	*HP-UX Version*	*Physical memory*	*File buffer cache size*	*Fixed storage*	*Users*	*Usage type*
cello	HP 9000/877	76	8.02	96 MB	10/30[b] MB	10.4 GB	20	Timesharing
snake	HP 9000/720	58	8.05	32 MB	5 MB	3.0 GB	200	Server
hplajw	HP 9000/845	23	8.00	32 MB	3 MB	0.3 GB	1	Workstation

a. Each machine uses an HP PA-RISC microprocessor.
b. Cello's file buffer size changed from 10MB to 30MB on April 26, 1992.

Table 1: The three computer systems traced

Disk	Formatted capacity	Track buffer	Cylinders	Data heads	Rotational speed	Average 8KB access	Host interconnect	
							type	speed
HP C2200A	335 MB	none	1449	8	4002 rpm	33.6 ms	HP-IB	1–1.2MB/s[a]
HP C2204A	1.3 GB	none	2x1449[b]	2x16	4002 rpm	30.9 ms	HP-FL	5MB/s
HP C2474S	1.3 GB	128 KB	1935	19	4002 rpm	22.8 ms	SCSI-II	5MB/s
HP 97560	1.3 GB	128 KB	1935	19	4002 rpm	22.8 ms	SCSI-II	10MB/s
Quantum PD425S	407 MB	56 KB	1520	9	3605 rpm	26.3 ms	SCSI-II	5MB/s

a. 1MB/s towards the disk, 1.2MB/s towards the host.
b. A C2204A disk has two 5.25" mechanisms made to look like a single disk drive by the controller.

Table 2: Disk details

The paper is organized as follows. We begin with a short overview of previous work in the area. Then comes a description of our tracing method and details of the systems we traced; it is followed by a detailed analysis of the I/O patterns observed on each of the systems. Then we present the results of both simulations of adding non-volatile write buffers in the disks, and conclude with a summary of our results.

Related work

Most I/O access pattern studies have been performed at the file system level of the operating system rather than at the disk level. Since logging every file system operation (particularly every read and write) generates huge quantities of data, most such studies have produced only summary statistics or made some other compromise such as coalescing multiple I/Os together (e.g., [Ousterhout85, Floyd86, Floyd89]). Many were taken on non-UNIX systems. For example: IBM mainframes [Procar82, Smith85, Kure88, Staelin88, Staelin90, Staelin91, Bozman91]; Cray supercomputers [Miller91]; Sprite (with no timing data on individual I/Os) [Baker91]; DEC VMS [Ramakrishnan92]; TOPS-10 (static analysis only) [Satyanarayanan81].

The UNIX buffer cache means that most accesses never reach the disk, so these studies are not very good models of what happens at the disk. They also ignore the effects of file-system generated traffic, such as for metadata and read-ahead, and the effects of swapping and paging. There have been a few studies of disk traffic, but most have had flaws of one kind or another. For example: poor measurement technology (60 Hz timer) [Johnson87]; short trace periods (75 minutes at a time, no detailed reporting of data, 2ms timer granularity) [Muller91]; limited use patterns [Carson90]. Raymie Stata had earlier used the same tracing technology as this study to look at the I/Os in a time-sharing UNIX environment [Stata90]. He discovered skewed device utilization and small average device queue lengths with large bursts.

We were interested in pursuing this path further, and gathering detailed statistics without the limitations of others' work. The next section details how we did so.

Trace gathering

We traced three different Hewlett-Packard computer systems (described in Table 1). All were running release 8 of the HP-UX operating system [Clegg86], which uses a version of the BSD fast file system [McKusick84]. The systems had several different types of disks attached to them, whose properties are summarized in Table 2.

Trace collection method

All of our data were obtained using a kernel-level trace facility built into HP-UX. The tracing is completely transparent to the users and adds no noticeable processor load to the system. We logged the trace data to dedicated disks to avoid perturbing the system being measured (the traffic to these disks is excluded from our study). Channel contention is minimal: the logging only generates about one write every 7 seconds.

Each trace record contained the following data about a single physical I/O:

- timings, to 1μs resolution, of enqueue time (when the disk driver first sees the request); start time (when the request is sent to the disk) and completion time (when the request returns from the disk);
- disk number, partition and device driver type;
- start address (in 1KB fragments);
- transfer size (in bytes);
- the drive's request queue length upon arrival at the disk driver, including the current request;
- flags for read/write, asynchronous/synchronous, block/character mode;
- the type of block accessed (inode, directory, indirect block, data, superblock, cylinder group bitmap)

The tracing was driven by a daemon spawned from init; killing the daemon once a day caused a new

snake						
ID	disk type	partition	size	number of I/Os		reads
A	Quantum PD425S	/ (root)	313 MB	5 848 567	46.6%	40%
		(swap)	94 MB	50 694	0.4%	91%
B	HP 97560	/usr1	1.3 GB	2 862 620	22.8%	48%
C	HP 97560	/usr2	1.3 GB	3 793 004	30.2%	43%
		Total	3.0 GB	12 554 885	100.0%	43%

hplajw						
ID	disk type	partition	size	number of I/Os		reads
A	HP 2200A	/ (root)	278 MB	346 553	83.0%	36%
		(swap)	24 MB	19 625	4.7%	71%
		(swap)	16 MB	17 940	4.3%	68%
B[a]	HP 2200A	(swap)	16 MB	31 113	7.5%	82%
		Total	334 MB	416 262	99.5%[b]	42%

a. The remaining portion of this drive is the area where the trace data was collected.
b. The percentages do not add up to 100% due to raw disk accesses to the boot partition on disk A.

cello						
ID	disk type	partition	size	number of I/Os		reads
A	HP C2474S	/ (root)	958 MB	3 488 934	11.9%	59%
		/usr/local/lib/news	105 MB	3 511 772	12.0%	44%
		(swap1)	126 MB	243 932	0.8%	80%
		(swap2)	48 MB	59 423	0.2%	70%
B	HP 2204A	/users	1.3 GB	4 162 026	14.2%	47%
C	HP 2204A	/usr/local/src	1.3 GB	1 027 978	3.5%	85%
D	HP 2204A	/usr/spool/news	1.3 GB	14 372 382	49.0%	36%
E	HP 2204A	/usr/spool/news/in.batches	105 MB	479 441	1.6%	13%
		/nfs/export/refdbms	29 MB	21 531	0.1%	84%
		/backup	946 MB	185 845	0.6%	22%
		/tmp	126 MB	623 663	2.1%	4%
		(swap3)	75 MB	212 324	0.7%	80%
F	HP 2204A	/nobackup	1.3 GB	668 687	2.3%	71%
G	HP 2204A	/nobackup-2	1.3 GB	68 431	0.2%	79%
H	HP 2204A	/mount/oldroot	1.3 GB	224 908	0.8%	88%
		Subtotal for swap partitions	249 MB	515 679	1.8%	79%
		Subtotal for the news partitions	1.4 GB	18 363 595	62.6%	37%
		Subtotal excluding news and swap	8.8 GB	10 472 003	35.6%	54%
		Grand total	10.4 GB	29 351 277	100.0%	44%

Table 3: Summary of the traces gathered; cello and hplajw were traced from 92.4.18 to 92.6.20; snake from 92.4.25 to 92.6.27

trace file to be started (the kernel's buffering scheme meant that no events were lost). Each time this happened, we also collected data on the physical memory size, the cache size, system process identifiers, mounted disks, and the swap configuration.

Traced systems

Cello is a timesharing system used by a small group of researchers at Hewlett-Packard Laboratories to do simulation, compilation, editing, and mail. A *news* feed that was updated continuously throughout the day resulted in the majority (63%) of the I/Os in the system, and these I/Os have a higher-than-usual amount of writes (63%). The other partitions vary, with the mean being 46% writes. Because of the large activity directed to the news partitions, the system as a whole does more writes (56%) than reads.

Snake acted as a file server for an HP-UX cluster [Bartlett88] of nine clients at the University of California, Berkeley. Each client was an Hewlett-Packard 9000/720 workstation with 24MB of main memory, 66MB of local swap space, and a 4MB file buffer cache. There was no local file system storage on any of the clients; all the machines in the cluster saw a single common file system with complete single-system semantics. The cluster had accounts for professors, staff, graduate students, and computer

science classes. The main use of the system was for compilation and editing. This cluster was new in January 1992, so many of the disk accesses were for the creation of new files. Over the tracing period, the */usr1* disk gained 243MB and */usr2* gained 120MB of data.

Finally, the personal workstation (hplajw) was used by the second author of this paper. The main uses of the system were electronic mail and editing papers. There was not much disk activity on this system: the file buffer cache was doing its job well.

Cello and hplajw were traced from 92.4.18 to 92.6.20; snake from 92.4.25 to 92.6.27. We also use a common week-long subset of the data for some analyses; this period runs from 92.5.30 to 92.6.6. All the numbers and graphs in this paper are derived from either the full or the week-long traces: we say explicitly if the shorter ones are being used. Each trace (full or short) starts at 0:00 hours on a Saturday.

The file system configurations for the three systems are given in Table 3. The total numbers of I/O requests logged over the tracing period discussed in this paper were: 29.4M (cello), 12.6M (snake) and 0.4M (hplajw).

The swap partitions are used as a backing store for the virtual memory system. In general, there is little activity (0.4% on snake, 1.8% on cello): these systems are reasonably well equipped with memory, or local swap space in the case of snake's diskless clients. The exception is hplajw, on which 16.5% of I/Os are for paging because of memory pressure from simultaneous execution of the X windowing system, FrameMaker, GNU Emacs, and a bibliography database program.

Analysis

This section presents our detailed analyses of the trace data. Although it represents measurements from a single file system design (the HP-UX/4.3BSD fast file system), we believe this data will be of use to other file system designers – particularly in providing upper bounds on the amount of disk traffic that might be saved by different approaches to designing file systems.

For example, we know that HP-UX is very enthusiastic about flushing metadata to disk to make the file system very robust against power failures.[1] This means that the metadata traffic we measured represents close to an upper bound on how much a metadata-logging file system might be able to suppress.

I/O timings

Figure 1 shows the distribution of both elapsed and physical I/O times for the three systems. The *physical time* is the time between the disk driver dispatching a request to the disk and the I/O completion event – i.e., approximately the time that the disk is busy; the *elapsed time* includes queueing delays. The values are shown in Figure 1 and Table 4. Large differences between these two times indicate that many I/Os are being queued up in the device driver waiting for previous I/Os to complete.

Typical causes for the difference in times include high system loads, bursty I/O activity, or an uneven distribution of load between the disks. Table 4 shows that the disparity in I/O times between elapsed and physical times is much larger for writes than for reads. This suggests that writes are very bursty. One cause is

[1] Other people might add "or crashes" here, but we've never experienced a system crash in 6 years of running HP-UX on over twenty PA-RISC machines.

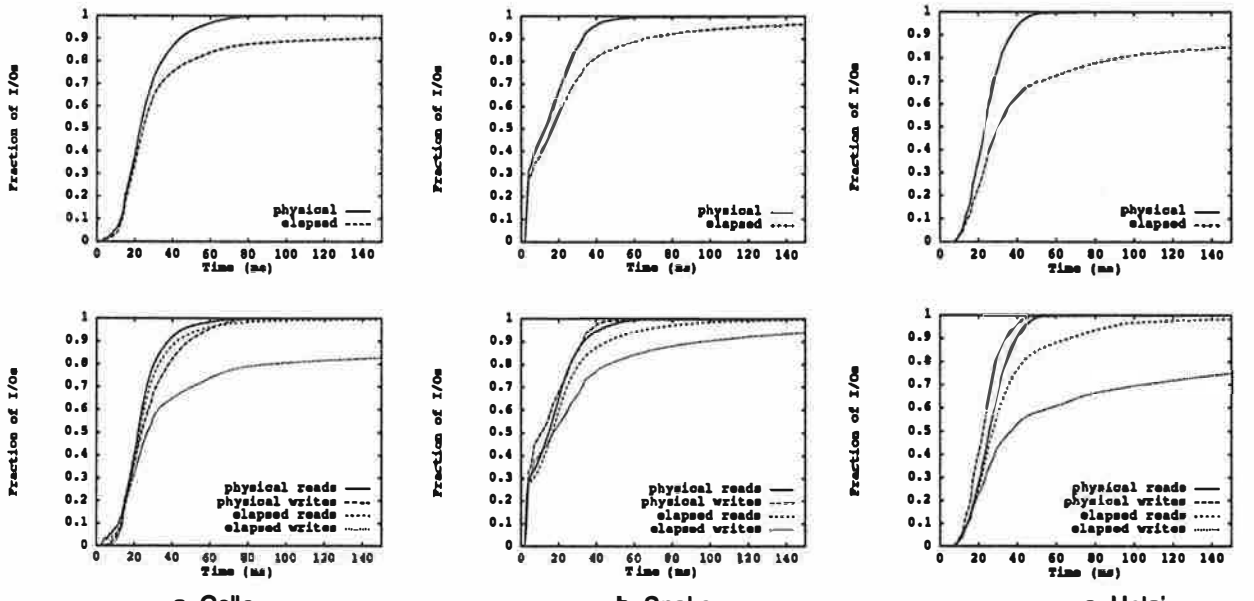
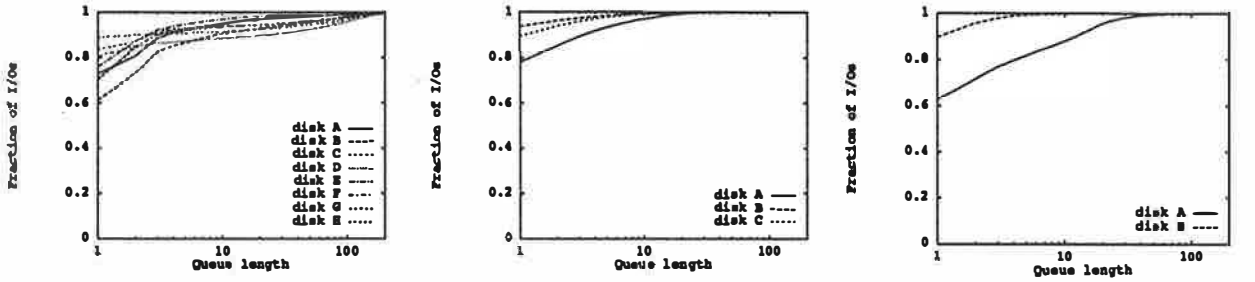


Figure 1: Distributions of physical and elapsed I/O times; see Table 4 for the mean values



a. Cello: mean = 8.9; stddev = 36.0 b. Snake: mean = 1.7, stddev = 3.5 c. Hplajw: mean = 4.1; stddev = 7.8

Figure 2: Queue length distributions for each disk in each system

the syncer daemon, which pushes delayed (buffered) writes out to disk every 30 seconds.

Snake has by far the fastest physical I/O times: 33% of them take less than 5ms. This is due to the use of disks with aggressive read-ahead and immediate write-reporting (the drive acknowledges some writes as soon as the data are in its buffer). More on this later.

The I/O loads on the disks on these systems are quite moderate – less than one request per second per disk. Nonetheless, the queue lengths can become quite large, as can be seen from Figures 2 and 3. Over 70% of the time, requests arrive at an idle disk. Some disks see queue lengths of 5 or more 15% of the time requests are added, and 5% of the time, experience queue lengths around 80 (cello) or 20 (hplajw). The maximum queue lengths seen are much larger: over 1000 requests on cello, over 100 on hplajw and snake. This suggests that improved request-scheduling algorithms may be beneficial [Seltzer90, Jacobson91].

The bursty nature of the arrival rate is also shown by Figure 4, which shows the overall arrival rates of requests, and by Figure 6, which shows request inter-arrival times. Many of the I/O operations are issued less than 20ms apart; 10–20% less than 1ms apart.

The I/O load on both snake and cello is significantly skewed, as Figure 5 shows: one disk receives most of the I/O operations on each system.

system	I/O type	physical	elapsed
cello	reads	23.6	27.4
	writes	27.7	272.0
	total	25.9	164.0
snake	reads	17.0	22.3
	writes	14.9	42.2
	total	15.8	33.7
hplajw	reads	27.5	39.2
	writes	24.1	142.0
	total	25.5	98.5

Table 4: Mean I/O request response times in ms

I/O types

Some previous studies (e.g., [Ousterhout85]) assumed that almost all accesses to the disk were for user data and therefore neglected to measure metadata and swap accesses. Our data (presented in Table 6)

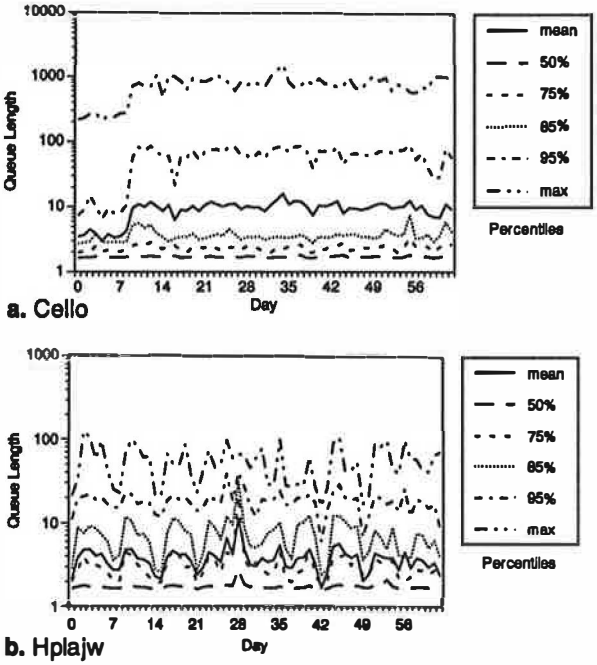
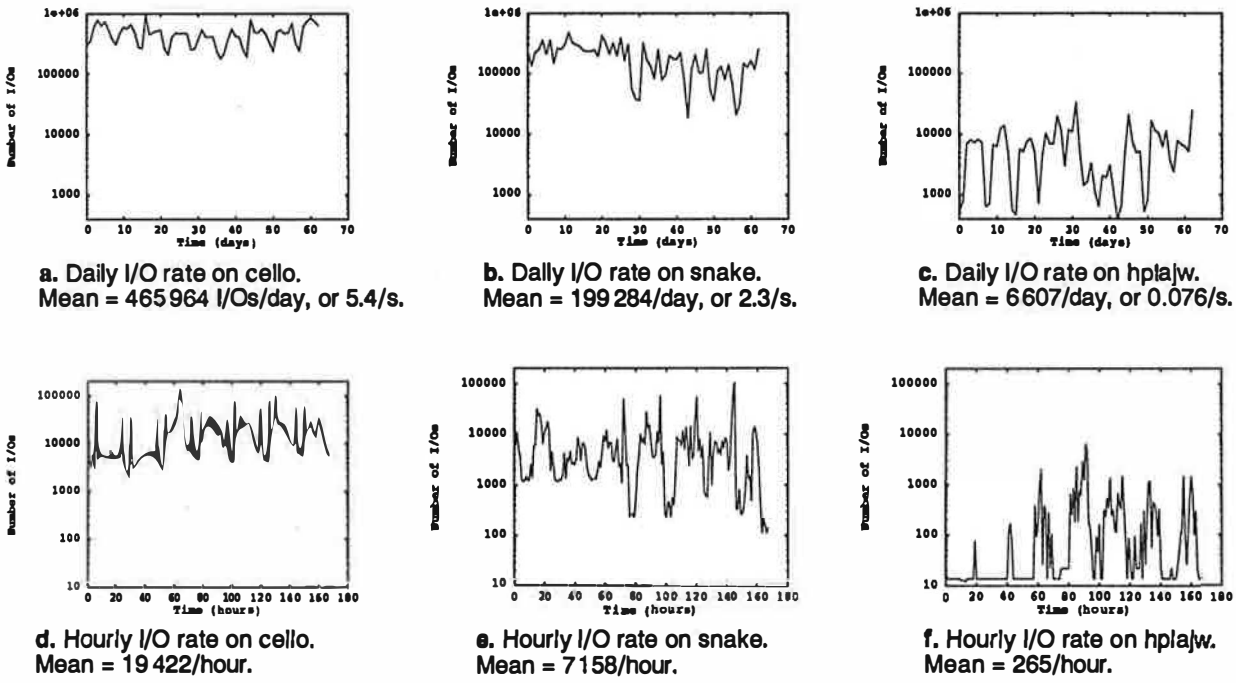


Figure 3: Mean queue length distributions versus time (daily values over 92.4.18–92.6.20)

percentile	cello			snake	hplajw
	A	B	D	A	A
% disk idle	73.0%	61.3%	70.5%	78.1%	63.0%
80%	2	3	2	2	5
90%	4	10	25	4	13
95%	11	43	89	7	20
99%	98	144	177	18	39
100%	606	1520	1070	111	124

Table 5: Per-disk queue length distributions



a. Daily I/O rate on cello.
Mean = 465 964 I/Os/day, or 5.4/s.

b. Daily I/O rate on snake.
Mean = 199 284/day, or 2.3/s.

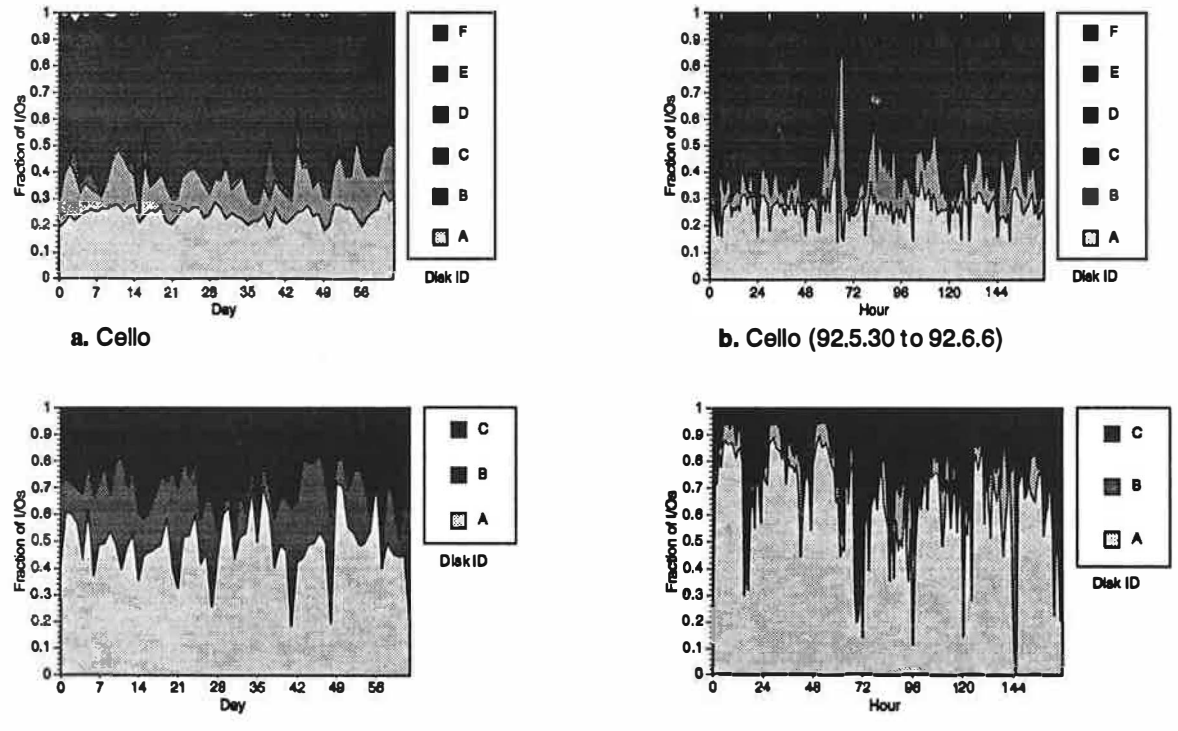
c. Daily I/O rate on hplajw.
Mean = 6607/day, or 0.076/s.

d. Hourly I/O rate on cello.
Mean = 19 422/hour.

e. Hourly I/O rate on snake.
Mean = 7158/hour.

f. Hourly I/O rate on hplajw.
Mean = 265/hour.

Figure 4: I/O rates for all three systems. All traces being on a Saturday; the hourly data spans 92.5.30–92.6.6



a. Cello

b. Cello (92.5.30 to 92.6.6)

c. Snake

d. Snake (92.5.30 to 92.6.6)

Figure 5: I/O load distribution across the disks for cello and snake. Disks **G** and **H** on cello are omitted, because they saw almost no I/O traffic; hplajw is omitted because almost all I/O was to the root disk.

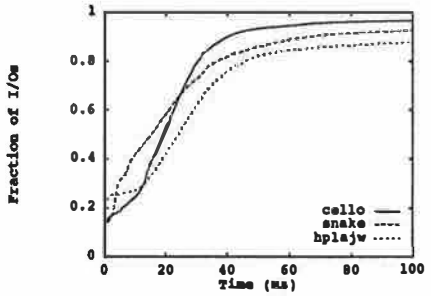
system	operation	I/O type				Raw disk I/Os
		user data	metadata	swap	unknown	
cello	reads	51.4% (66%)	40.0% (100%)	3.2% (100%)	5.5% (100%)	8.6%
	writes	32.0% (55%)	67.3% (72%)	0.7% (100%)	0.0% (100%)	0.7%
	total	40.6% (61%)	55.2% (81%)	1.8% (100%)	2.4% (100%)	4.2%
snake	reads	5.7% (86%)	17.4% (100%)	0.7% (100%)	76.2% (83%)	34.0%
	writes	19.5% (46%)	78.1% (18%)	0.1% (100%)	2.3% (45%)	0.1%
	total	13.4% (53%)	51.5% (30%)	0.4% (100%)	34.7% (82%)	15.0%
hplajw	reads	23.6% (60%)	10.7% (100%)	29.2% (100%)	36.4% (100%)	65.7%
	writes	21.7% (26%)	71.2% (52%)	7.1% (100%)	0.0% (100%)	7.1%
	total	22.5% (41%)	45.5% (57%)	16.5% (100%)	15.4% (100%)	31.9%

Table 6: distribution of I/Os by type

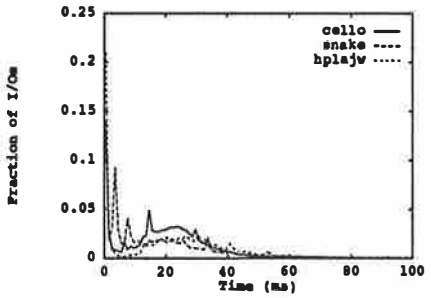
In this table, *user data* means file contents, *metadata* includes inode, directory, indirect blocks, superblock, and other bookkeeping accesses, *swap* corresponds to virtual memory and swapping traffic, and *unknown* represents blocks classified as such by the tracing system (they are demand-paged executables and, on snake, diskless-cluster I/Os made on behalf of its clients). The percentages under "I/O type" sum to 100% in each row.

The amount of raw (character-mode, or non-file-system) traffic is also shown as a percentage of the entire I/Os. Raw accesses are made up of the swap traffic and the demand-paging of executables from the *unknown* category. On hplajw, there was also a small amount of traffic to the boot partition in this mode.

Numbers in parentheses represent the percentage of that kind of I/O that was synchronous at the file system level (i.e., did not explicitly have the *asynchronous-I/O* flag attached to the I/O request).



a. inter-arrival distributions



b. inter-arrival densities

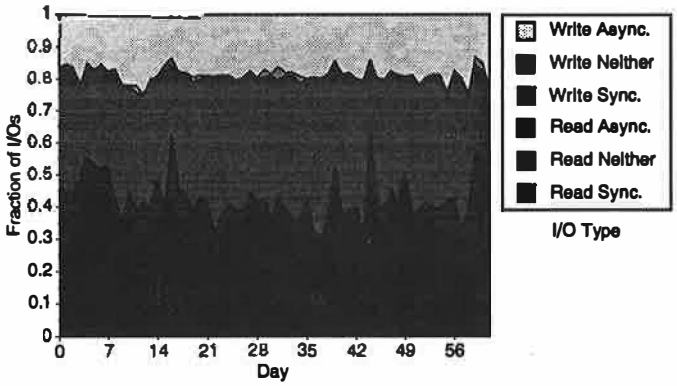
Figure 6: System-wide distribution and density plots of I/O request inter-arrival times. Cello mean: 185ms; snake mean: 434ms; hplajw mean: 13072ms

suggest that these assumptions are incorrect. In every trace, the user data is only 13–41% of all the I/Os generated in each system. The metadata percentage is especially high for writes, where 67–78% of the writes are to metadata blocks.

The ratio of reads to writes varies over time (Figure 8). The spikes in the hourly graph correspond to nightly backups. Overall, reads represent 44% (cello), 43% (snake) and 42% (hplajw) of all disk accesses. This is a surprisingly large fraction, given the sizes of the buffer caches on these systems – cello in particular. When cello's buffer cache was increased from 10MB to 30MB, the fraction of reads declined from 49.6% to 43.1%: a surprisingly small reduction. This suggests that predictions that almost all reads would be absorbed by large buffer caches [Ousterhout85, Ousterhout89, Rosenblum92] may be overly optimistic.

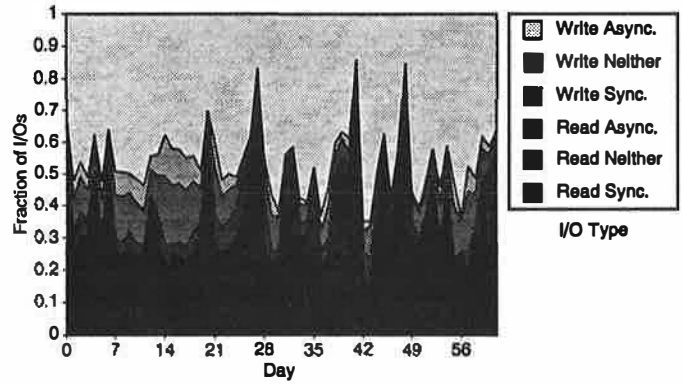
The HP-UX file system generates both *synchronous* and *asynchronous* requests. Synchronous I/O operations cause the invoking process to wait until the I/O operation has occurred before proceeding, so delaying synchronous operations can increase the latency associated with an I/O operation. Asynchronous operations proceed in the background, not tied directly to any process. Requests not explicitly flagged either way are treated as synchronous at the file system level, asynchronous at the disk level (this distinction is only important for writes).

Most read requests are implicitly synchronous (Table 6 and Figure 7), except for user data, where 14–



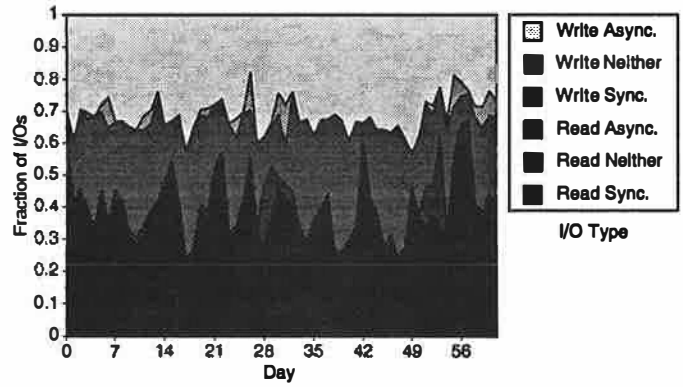
a. Cello

Mean values	reads	writes	both
asynchronous	7.7%	18.7%	26.4%
neither	36.4%	0.8%	37.2%
synchronous	0.0%	36.4%	36.4%
total	44.1%	55.9%	100.0%



b. Snake

Mean values	reads	writes	both
asynchronous	5.9%	42.6%	48.5%
neither	37.9%	4.6%	42.5%
synchronous	0.0%	9.0%	9.0%
total	43.8%	56.2%	100.0%



c. Hplajw

Mean values	reads	writes	both
asynchronous	4.0%	28.8%	32.8%
neither	38.4%	4.4%	42.8%
synchronous	0.0%	24.4%	24.4%
total	42.4%	57.6%	100.0%

Figure 7: I/Os on each system classified by type, expressed as a fraction of the total I/Os on that system.The labels *synchronous* and *asynchronous* indicate that one of these flags was associated by the file system with the request; *neither* indicates that the file system did not explicitly mark the request in either way. The flags do not occur together.

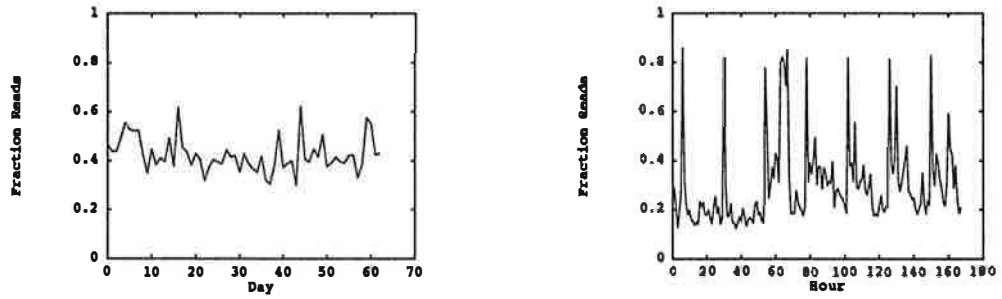


Figure 8: Fraction of I/Os on cello that are reads as a function of time (daily: 92.4.18 to 92.6.20; hourly: 92.5.30 to 92.6.6). Note the reduction in read fraction after cello's buffer cache was increased in size on 92.4.26.

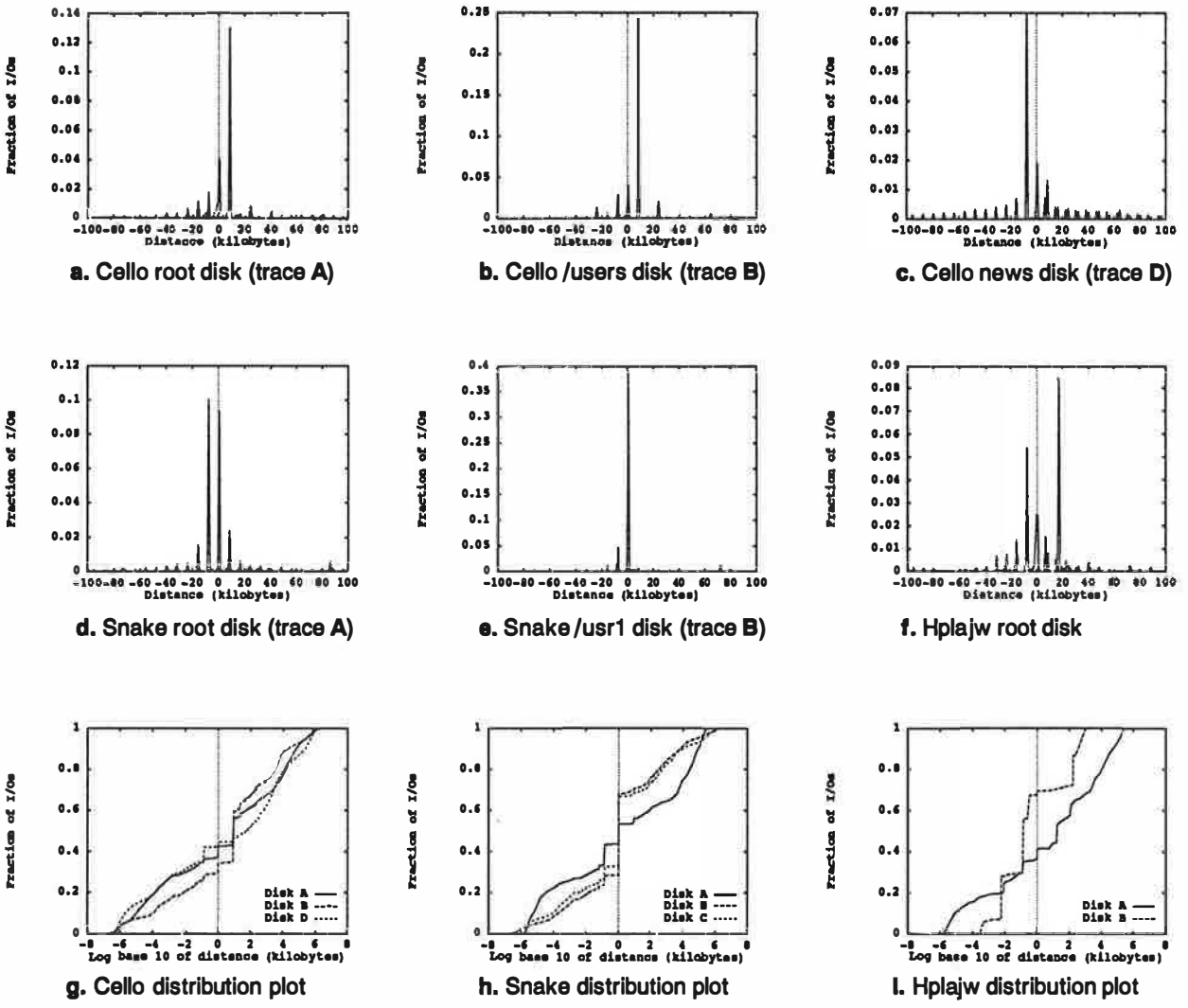


Figure 9: Density and distribution plots of the distance in KB between the end of one request and the start of the next. In the distribution plots, the X-axis is given by $x = \text{sign}(d) \times \log_{10}(|d|)$ of the distance d. The large peaks at –8 KB correspond to block overwrites. The traces run from 92.5.30 to 92.6.6.

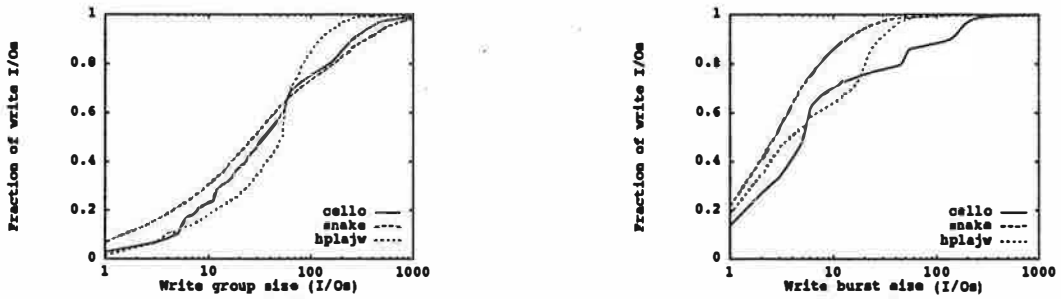
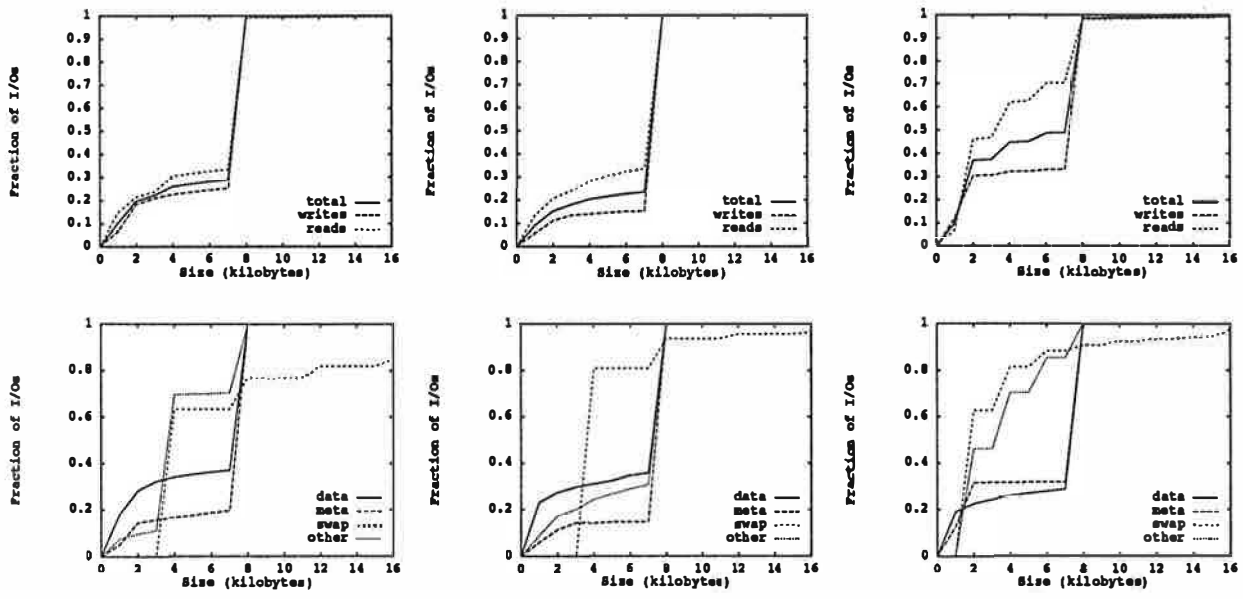


Figure 10: Distributions of *per-disk* write group sizes and write burst sizes



a. Cello b. Snake c. Hplajw

Figure 11: Distribution of transfer sizes for all systems

40% of the accesses are asynchronous read-aheads. Writes are almost all explicitly flagged as synchronous or asynchronous; again, it is user data that has the highest amount (45–74%) of asynchronous activity, except on snake, where almost all metadata writes (82%) are asynchronous. All swap and paging traffic is synchronous.

Write groups and write bursts

Looking just at the stream of write requests, we found that many of the writes occurred in *groups*, with no intervening reads. Writes rarely occur on their own: Figure 10 shows that only 2–10% of writes occur singly; 50% of them happen in groups of 20 or more (snake) to 50 or more (hplajw). Write *bursts* are almost as common. (A burst includes all those I/Os that occur closer than 30ms to their predecessor.) Most writes (60–75%) occur in bursts. On cello, 10% of the writes occur in bursts of more than 100 requests.

I/O placement

Previous studies have shown that I/Os at the file system level are highly sequential [Ousterhout85, Baker91]. But our data (plotted in Figure 9) shows that by the time these requests reach the disk they are much less so.

We define requests to be *logically sequential* if they are at adjacent disk addresses or disk addresses spaced by the file system interleave factor. There is a wide range of logical sequentiality: from 4% (on the cello news disk), 9% (hplajw root disk) to 38% (snake /usr1 disk). The means for the three systems are shown in Table 7, expressed as percentages of all I/Os.

system	reads	writes
cello	7.7%[a]	6.3%
snake	14.1%	4.7%
hplajw	4.5%	5.0%

a. 15.4% without the news disk.

Table 7: Fraction of I/Os that are logically sequential

I/O sizes

An I/O device can be accessed either through the block device or the character device. When a file is accessed via the block device, all I/O is done in multiples of the fragment size (1KB) up to the block size (8KB). Disks accessed via the character device (e.g., for demand paging of executables or swapping) have no such upper bound, although they are always multiples of the machine's physical page size: 2KB for hplajw, 4KB for the other two.

As Table 6 shows, most all accesses go through the file system, except on hplajw, where there is a large amount of swap and paging traffic (32% of the requests). Figure 11 shows how the distribution of I/O sizes varies across the systems we traced as a function of the kind of I/O being performed. As expected, file system writes are up to 8KB in size, while swap accesses can be larger than this.

Block overwrites

We observed a great deal of block overwriting: the same block (typically a metadata block) would be written to disk over and over again. One cause of this is a file that is growing by small amounts: each time the

file is extended, HP-UX posts the new inode metadata to disk – metadata is essentially held in a write-through cache.

Figure 12 plots the time between overwrites of the same block. On the root disks, 25% (hplajw and cello) to 38% (snake) of updated blocks are overwritten in less than 1 second; 45% of the blocks are overwritten in 30 seconds (cello); 18% of the blocks are overwritten at 30-second intervals (snake – presumably the syncer daemon); and over 85% of all blocks written are overwritten in an hour or less – 98% for snake.

A similar picture is told by the block access distributions shown in Figure 13. Up to 30% of the writes are directed to just 10 blocks, and 65–100% of the writes go to the most popular 1000 blocks; 1% of the blocks receive over 90% of the writes.

Together, these figures suggest that caching only a small percentage of the blocks in non-volatile memory could eliminate a large fraction of the overwrites.

Immediate reporting

The disks on snake use a technique known as *immediate reporting* for some of their writes. Our studies show that enabling it reduces the mean write time from 20.9ms to 13.2ms.

HP-UX's immediate reporting is intended to provide faster write throughput for isolated writes and sequential transfers. It operates as follows. An *eligible* write command is acknowledged as complete by the disk drive as soon as the data has reached the drive's volatile buffer memory. Eligible writes are those that are explicitly or implicitly asynchronous, and those that are physically immediately after the write that the drive is currently processing. The disk's write buffer acts solely as a FIFO: no request reordering occurs.

Since the data that is immediately-reported is vulnerable to power failures until it is written to disk, HP-UX disables immediate reporting for write requests explicitly flagged as synchronous.

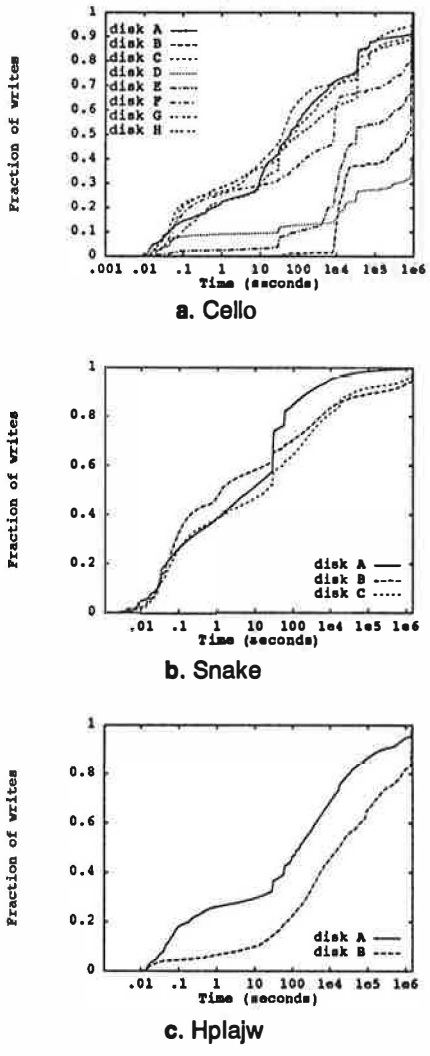


Figure 12: Distribution of 8KB-block overwrite delays

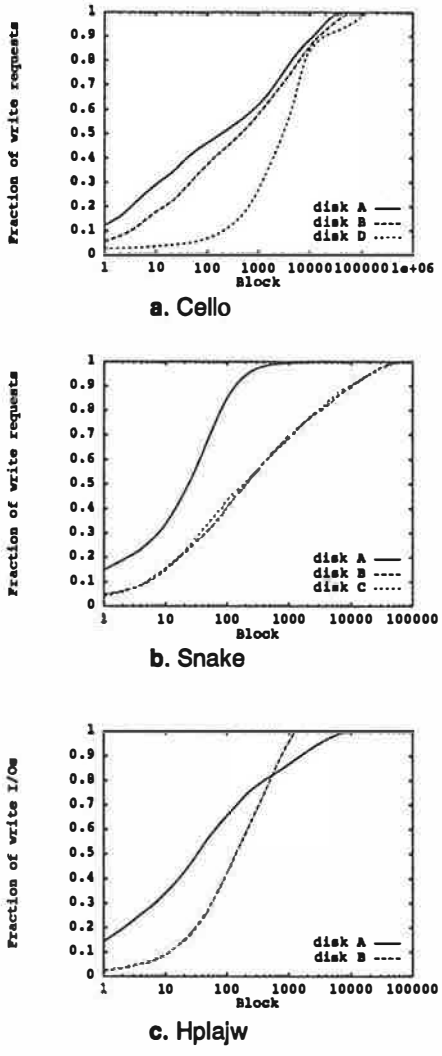


Figure 13: Distribution of writes by 8KB block number; blocks are sorted by write access count

When immediate reporting is enabled, writes are faster, and take only 3–7ms for an 8KB block. Figure 1 shows that 45% of all writes occur in less than 7ms. This is 53% of the writes eligible for immediate reporting (determined from Figure 7 and Table 3). The minimum physical request time is around 3ms, made up of 1.6ms of SCSI bus data transfer time, plus about the same again in disk, driver and SCSI channel overheads.

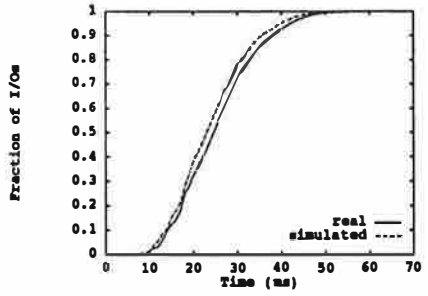
The main benefit of immediate reporting is that sequential, back-to-back writes can proceed at full disk data rates, without the need for block interleaving. (On the HP97560 disks on snake, this means 2.2 MB/sec.) However, only 4.7–6.3% I/Os are sequential writes, so the benefit is not as great as might be hoped. Perhaps caching, which allows request reordering in the disk, could help alleviate this problem. To investigate this, we turned to simulation studies driven by our trace data.

Simulation studies

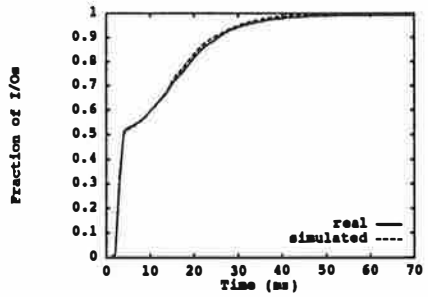
We constructed a simulation model to explore various design alternatives in the I/O subsystem. We report here on our results with adding non-volatile RAM (NVRAM) caches. We begin with a description of the model itself, and the calibration we performed on it. We then present the results of applying different caching strategies.

The simulation model

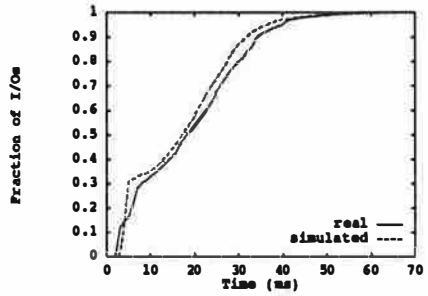
We modelled the disk I/O subsystem in (excruciating) detail, including transfer times over the SCSI bus from the host, bus contention, controller overhead, seek times as a (measured) function of distance, read and write settling times, rotation position, track- and cylinder-switch times, zone bit recording for those disks that had it, media transfer rate, and placement of sparing areas on the disks.



a. HP C2200A

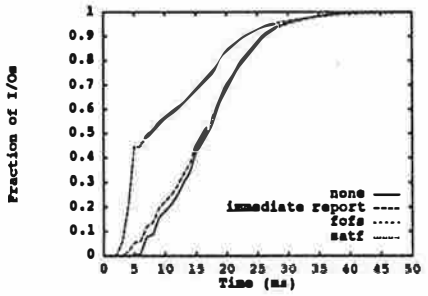


b. HP97560

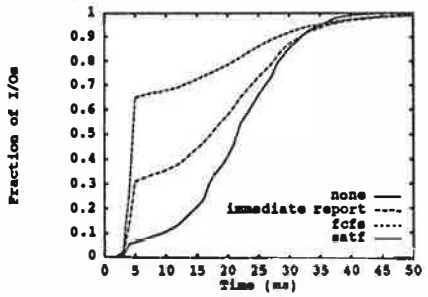


c. Quantum PD425S

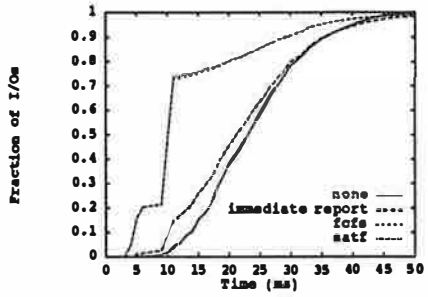
Figure 14: Measured and modelled physical disk I/O times over the period 92.5.30–92.6.6



a. Cello root disk



b. Snake root disk



c. Hplajw root disk

Figure 15: Distributions of physical I/O times for different disk caching policies with 128KB of cache, over the period 92.5.30–92.6.6

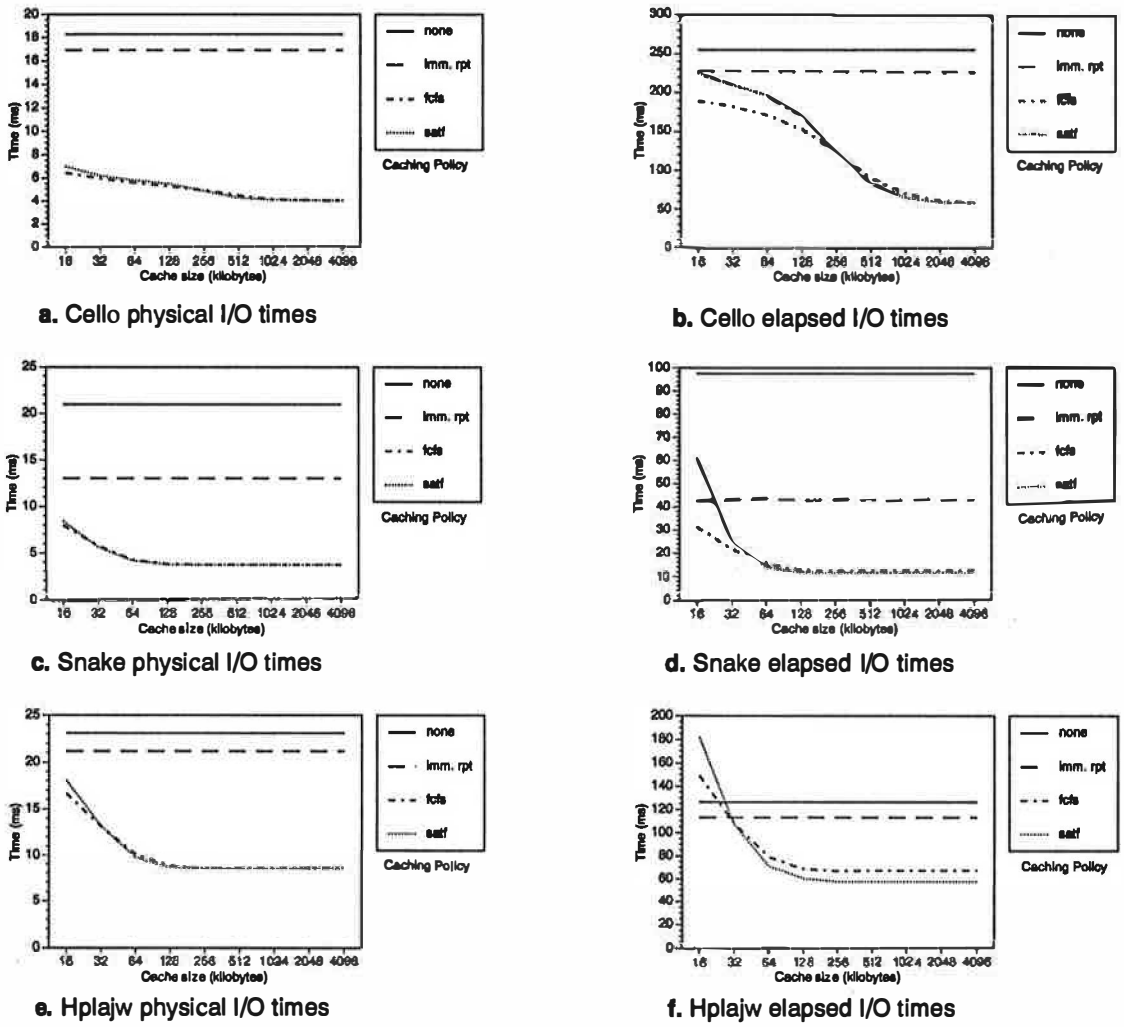


Figure 16: Physical and elapsed times for different cache sizes and caching policies. The root disk from each system is shown; traces are from 92.5.30–92.6.6.

To calibrate our model, we compared the measured I/O times in the traces against the three disks we simulated. The result is the close match shown in Figure 14. We did not attempt to mimic the two-spindle, single-controller HP2204A: instead, we modelled it as an HP2474S (but with read-ahead disabled, since the HP2204 controllers do not provide it). Since our results compare simulator runs, rather than compare simulations against the real trace (other than for the calibration), we believe the results are still useful.

Read-ahead at the disk

We did a limited exploration of the effects of in-disk read-ahead. If read-ahead is enabled, the disk continues to read sequentially into an internal read-ahead cache buffer after a read until either a new request arrives, or the buffer fills up. (This buffer is independent of the write caches we discuss later.) In the best case, sequential reads can proceed at the full disk transfer rate. The added latency for other requests can be made as small as the time for a single sector read (0.2ms for an HP97560) in all but one case: if the read-ahead crosses a track boundary, the track switch proceeds to completion even if there is a new request to service.[2]

Table 8 shows the effects of disabling or enabling read-ahead for the cello traces. Enabling it improves physical read performance by 10% and elapsed read times by 42%, but has no effect on write times.

[2] A more recent HP disk, the HP C3010, lets the host decide whether such inter-track read-aheads should occur.

read-ahead?	Reads		Writes	
	elapsed	physical	elapsed	physical
no	33.7ms	16.4ms	255.4ms	18.3ms
yes	19.5ms	14.7ms	255.5ms	18.3ms

Table 8: Effect of read-ahead on I/O times, averaged over all disks on cello; 92.5.30–92.6.6

Non-volatile write caching at the disk

If non-volatile memory were used for the write buffers in the disk, the limitations of immediate reporting could be lifted: both synchronous and non-sequential writes could be cached. In particular, we were interested in how the write-back policy from the cache would affect performance.

Policies we explore here include: no cache at all, immediate reporting, caching with a straight FCFS scheduling algorithm, and caching with a modified shortest access time first scheduling (*SATF*) algorithm. (SATF is a scheduling algorithm that takes both seek and rotation position into account [Seltzer90, Jacobson91]. We modified it to favor writing large blocks out of the cache over small ones since this gave better performance at small cache sizes: it freed up space in the cache faster.)

We gave reads priority over flushing dirty buffers from the cache to disk, given the small number of asynchronous reads we saw in our traces. Each simulated disk was also given a reserved buffer for reads so that these did not have to wait for space in the write buffer. In addition, large writes (>32KB) were streamed straight to the disk, bypassing the write buffer.

The results are presented in Figure 15 and 16, which show how the I/O times change under the different policies for the traces from the different systems.

We were surprised by two things in this data: first, there was almost no difference in the mean physical I/O times between the FCFS and SATF scheduling disciplines in the disk. In this context, FCFS is really the SCAN algorithm used by the device driver (modified by overwrites, which are absorbed by the cache). With small numbers of requests to choose from, SATF may second-guess the request stream from the host – and get it wrong – in the face of incoming read requests. At larger cache sizes, this effect is compensated for by the increased number of requests SATF can select from.

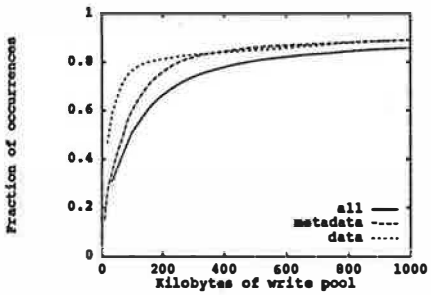
Second, even though the mean physical times were uniformly better when caching was enabled, the elapsed times for small cache sizes were sometimes worse. We tracked this down to the cache buffer replacement policy: the cache slots occupied by a request are not freed until the entire write has finished, so that an incoming write may have to wait for the entire current write to complete. At small cache sizes, this has the effect of increasing the physical times of writes that happen in bursts big enough to fill the cache – thereby accentuating the queueing delays, which occur mostly in these circumstances.

We also found that a very small amount of NVRAM (even as little as 8KB per disk) at the SCSI controller or in the host would eliminate 10–18% of the write traffic, as a result of the many overwrites. Indeed, on snake, 44–67% of metadata writes are overwritten in a 30 second period: absorbing all of these would reduce the total I/Os by 20%.

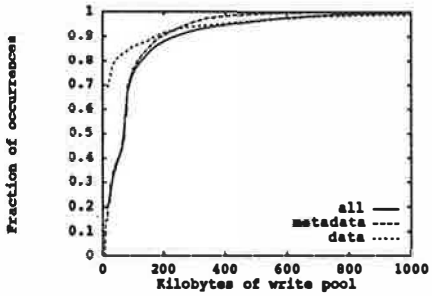
Reads were slowed down a little (less than 4%) when caching was turned on. We surmise that the increased cache efficiency increased the mean seek distance for writes by absorbing many of the overwrites. This meant that a read issued while a cache-flush write was active would occasionally have to wait for a slightly longer seek to complete than it would have done if there had been no caching. Also, reads do not interrupt a write: if this happens, the physical read time will include the time for the write to finish.

Non-volatile write caching at the host

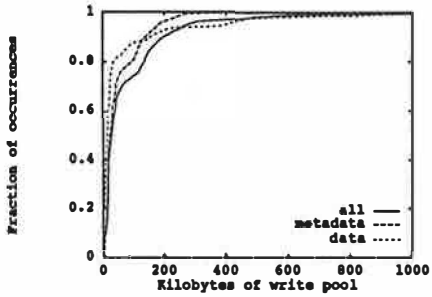
We then determined how the amount of NVRAM affected how many writes we could absorb in each 30 second interval in our traces. (We assumed no background write activity to empty the cache during the 30 second intervals.) We show this data in two ways: Figure 17 shows the distribution of 30-second intervals



a. Cello (92.5.30–92.6.6)



b. Snake (92.5.30–92.6.6)



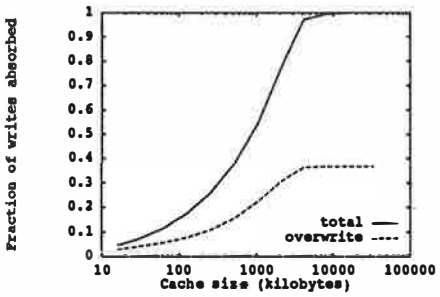
c. Hplajw (92.5.30–92.6.6)

Figure 17: Distributions of the cache sizes needed to absorb all write accesses over 30 second intervals.

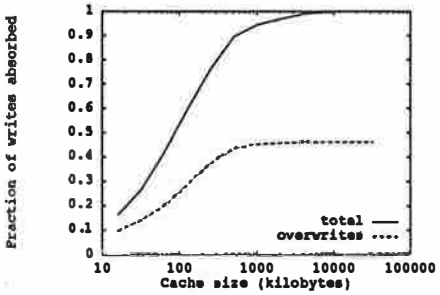
in which a given amount of cache was able to absorb *all* of the writes; Figure 18 shows the fraction of writes absorbed by a given cache size. (The disparity between the "overwrites" and "total" lines in the latter represents the amount of valid data in the NVRAM cache at the end of the 30 second period.)

Two hundred KB of NVRAM can absorb all writes in 65% (cello) to 90% (hplajw) of 30 second intervals. If metadata alone is considered (because it has the highest percentage of synchronous I/Os), all metadata writes can be absorbed in 80% of the intervals with 100KB (hplajw and snake) to 250KB (cello) of NVRAM.

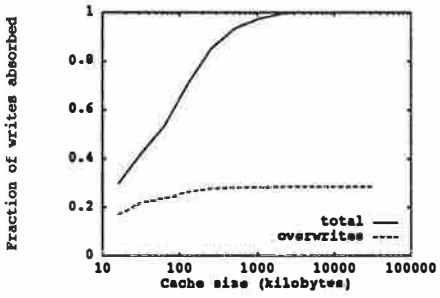
The "total" lines in Figure 18 show the write I/O bandwidth: once the cache is big enough to absorb this, little further improvement is seen. 95% absorption is reached at 700KB (hplajw), 1MB (snake) and 4MB (cello). Overwrites account for 25% (hplajw) to 47% (snake) of all writes.



a. Cello (92.5.30–92.6.6)



b. Snake (92.5.30–92.6.6)



c. Hplajw (92.5.30–92.6.6)

Figure 18: distributions of the number of writes absorbed by given cache sizes over 30 second intervals.

Conclusions

We have provided a great deal of information on three complete, disk-level I/O traces from computer systems with moderately disparate workloads. These results will be of use to others in understanding what file systems do, to evaluate possible changes, and to provide distribution parameters for modelling.

We have also presented the results of simulations of write caching at the disk level, and demonstrated that this is an effective technique, although a new finding is that the write scheduling policy has little effect on the cache efficacy.

Acknowledgments

This work was carried out as part of the DataMesh research project at HP Laboratories. We thank the users and administrators of the systems we traced for their cooperation in allowing us to gather the data. David Jacobson helped us interpret some of our results.

Availability

For researchers wishing greater detail than can be reproduced here, we have made the raw data for the graphs in this paper available via anonymous *ftp* from ftp.hpl.hp.com, in the file pub/wilkes/USENIX.Jan93.tar.

References

[Baker91] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. *Proceedings of 13th ACM Symposium on Operating Systems Principles* (Asilomar, Pacific Grove, CA). Published as *Operating Systems Review* **25**(5):198–212, 13–16 October 1991.

[Bartlett88] Debra S. Bartlett and Joel D. Tesler. A discless HP-UX file system. *Hewlett-Packard Journal* **39**(5):10–14, October 1988.

[Bozman91] G. P. Bozman, H. H. Ghannad, and E. D. Weinberger. A trace-driven study of CMS file references. *IBM Journal of Research and Development* **35**(5/6):815–28, Sept.–Nov. 1991.

[Carson90] Scott D. Carson. Experimental performance evaluation of the Berkeley file system. Technical report UMIACS–TR–90–5 and CS–TR–2387. Institute for Advanced Computer Studies, University of Maryland, January 1990.

[Clegg86] Frederick W. Clegg, Gary Shiu-Fan Ho, Steven R. Kusmer, and John R. Sontag. The HP-UX operating system on HP Precision Architecture computers. *Hewlett-Packard Journal* **37**(12):4–22, December 1986.

[English92] Robert M. English and Alexander A. Stepanov. Loge: a self-organizing storage device. *USENIX Winter 1992 Technical Conference Proceedings* (San Francisco, CA), pages 237–51, 20–24 January 1992.

[Floyd86] Rick Floyd. Short-term file reference patterns in a UNIX environment. Technical report 177. Computer Science Department, University of Rochester, NY, March 1986.

[Floyd89] Richard A. Floyd and Carla Schlatter Ellis. Directory reference patterns in hierarchical file systems. *IEEE Transactions on Knowledge and Data Engineering* **1**(2):238–47, June 1989.

[Jacobson91] David M. Jacobson and John Wilkes. Disk scheduling algorithms based on rotational position. Technical report HPL–CSP–91–7. Hewlett-Packard Laboratories, 24 February 1991.

[Johnson87] Thomas D. Johnson, Jonathan M. Smith, and Eric S. Wilson. Disk response time measurements. *USENIX Winter 1987 Technical Conference Proceedings* (Washington, DC), pages 147–62, 21–23 January 1987.

[Kure88] Øivind Kure. *Optimization of file migration in distributed systems*. PhD thesis, published as UCB/CSD 88/413. Computer Science Division, Department of Electrical Engineering and Computer Science, UC Berkeley, April 1988.

[McKusick84] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems* **2**(3):181–97, August 1984.

[Miller91] Ethan L. Miller and Randy H. Katz. Analyzing the I/O behavior of supercomputer applications. *Digest of papers, 11th IEEE Symposium on Mass Storage Systems* (Monterey, CA), pages 51–9, 7–10 October 1991.

[Muller91] Keith Muller and Joseph Pasquale. A high performance multi-structured file system design. *Proceedings of 13th ACM Symposium on Operating Systems Principles* (Asilomar, Pacific Grove, CA). Published as *Operating Systems Review* **25**(5):56–67, 13–16 October 1991.

[Ousterhout85] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. *Proceedings of 10th ACM Symposium on Operating Systems Principles* (Orcas Island, WA). Published as *Operating Systems Review* **19**(5):15–24, December 1985.

[Ousterhout89] John Ousterhout and Fred Douglis. Beating the I/O bottleneck: a case for log-structured file systems. *Operating Systems Review* **23**(1):11–27, January 1989.

[Porcar82] Juan M. Porcar. *File migration in distributed computer systems*. PhD thesis, published as Technical report LBL–14763. Physics, Computer Science and Mathematics Division, Lawrence Berkeley Laboratory, UC Berkeley, July 1982.

[Ramakrishnan92] K. K. Ramakrishnan, Prabuddha Biswas, and Ramakrishna Karedla. Analysis of file I/O traces in commercial computing environments. *Proceedings of 1992 ACM SIGMETRICS and PERFORMANCE92 International Conference on Measurement and Modeling of Computer Systems* (Newport, RI). Published as *Performance Evaluation Review* **20**(1):78–90, 1–5 June 1992.

[Rosenblum92] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, **10**(1):26–52, February 1992.

[Satyanarayanan81] M. Satyanarayanan. A study of file sizes and functional lifetimes. *Proceedings of 8th ACM Symposium on Operating Systems Principles* (Asilomar, Ca). Published as *Operating Systems Review*, **15**(5):96–108, December 1981.

[Seltzer90] Margo Seltzer, Peter Chen, and John Ousterhout. Disk scheduling revisited. *USENIX Winter 1990 Technical Conference Proceedings* (Washington, DC), pages 313–23, 22–26 Jan. 1990.

[Smith85] Alan Jay Smith. Disk cache—miss ratio analysis and design considerations. *ACM Transactions on Computer Systems* **3**(3):161–203, August 1985.

[Staelin88] Carl Staelin. File access patterns. Technical report CS–TR–179–88. Department of Computer Science, Princeton University, September 1988.

[Staelin91] Carl Staelin and Hector Garcia-Molina. Smart filesystems. *USENIX Winter 1991 Technical Conference Proceedings* (Dallas, TX), pages 45–51, 21–25 January 1991.

[Stata90] Raymie Stata. *File systems with multiple file implementations*. Masters thesis, published as a technical report. Dept of Electrical Engineering and Computer Science, MIT, 22 May 1990.

Author information

Chris Ruemmler is currently finishing his MS degree in Computer Science at the University of California, Berkeley. He received his BS degree from the University of California, Berkeley in May, 1991. He completed the work in this paper during an internship at Hewlett-Packard Laboratories. His technical interests include architectural design, operating systems, graphics, and watching disks spin around and around at 4002 RPM. His personal interests include deep sea fishing, swimming, and music.

John Wilkes graduated with degrees in Physics (BA 1978, MA 1980), and a Diploma (1979) and PhD (1984) in Computer Science from the University of Cambridge. He has worked since 1982 as a researcher and project manager at Hewlett-Packard Laboratories. His current research interests are generally in the area of resource management in scalable systems, and in particular in fast, highly available parallel storage systems. He particularly enjoys working with university students on projects such as this one.

The authors can be reached by electronic mail at ruemmler@cs.berkeley.edu and wilkes@hpl.hp.com.

An Analysis of File Migration in a UNIX Supercomputing Environment

Ethan L. Miller & Randy H. Katz – University of California, Berkeley

ABSTRACT

The supercomputer center at the National Center for Atmospheric Research (NCAR) migrates large numbers of files to and from its mass storage system (MSS) because there is insufficient space to store them on the Cray supercomputer's local disks. This paper presents an analysis of file migration data collected over two years. The analysis shows that requests to the MSS are periodic, with one day and one week periods. Read requests to the MSS account for the majority of the periodicity, as write requests are relatively constant over the course of a week. Additionally, reads show a far greater fluctuation than writes over a day and week since reads are driven by human users while writes are machine-driven.

1 Introduction

Over the last decade, computers have made incredible gains in speed. This speedup has encouraged the processing of larger and larger amounts of data; however, storing this data on magnetic disk is not feasible. Instead, most data centers with large data sets use tertiary storage devices such as tapes and optical disks to store much of their data. These devices provide a lower cost per megabyte of storage but have longer access times than magnetic disk. By studying the tradeoffs between cheaper and slower tertiary storage and more expensive and faster disk storage, response time can be improved without increasing storage costs.

The problem is especially acute at computer centers, such as the National Center for Atmospheric Research (NCAR), that deal with large amounts of data that can never be deleted. Data grows at the rate of several terabytes per year [20]. The cost of storing this data on shelved magnetic tape is relatively low, as cartridge tapes are inexpensive. However, storing even 1% of the total data in magnetic disk would be expensive, requiring hundreds of gigabytes of Cray disk storage.

This paper analyzes file migration behavior on the NCAR system described in [1] and [18]. The first section will provide some background on the problem, discussing current mass storage systems and previous work on them. The next section will describe the NCAR system in more detail. We will then present our trace-gathering methods.

The main part of the paper is a two-part analysis of the gathered trace data-analyzing the usage patterns for the entire mass storage system (MSS), and studying the behavior of individual files. The first part of the analysis includes system behavior over the course of a day, week, and longer periods. It characterizes user behavior with respect to the entire MSS, showing at what rate data and files are read and written. Other characteristics of the mass store at NCAR, such as request latency and interrequest distribution, are also discussed. The second part of the analysis provides insight for designing migration algorithms, as it focuses on how individual files are treated. This part of the analysis will discuss file size distribution and individual file reference patterns.

We will finally present some implications of our findings on migration algorithms, and suggest some directions for future research.

2 Background

History

File migration systems are used by many large computer installations, such as NCAR [1,18] and NASA [7,19], to store more data than what would cost-effectively fit on magnetic disk. Tertiary storage, which usually consists of tape and optical disk, lies at the bottom of the "storage pyramid," as shown in Figure 1. Cost and speed increase going up the pyramid, while the size of the memory level increases towards the bottom of the period. CPU cache is at the top of the pyramid; it have the highest cost per byte and is the smallest and fastest of the levels. At the bottom of the pyramid are tape and optical disk, which have slow access speeds, on the order of seconds or minutes, and very low cost, under $10/GB.

Early mass storage systems used manual tape mounting, since it was cheaper to hire system operators than it was to have a robot manage tape mounts. However, by 1978, several companies had introduced automated tape systems [2], and automated tape storage became part of the mass storage systems in most large computer centers. Several of these centers were studied in the early 1980s; these included Brookhaven National Laboratory [5], the University of Illinois [10], and the Stanford Linear Accelerator Center [14,15]. These will be discussed in a later section.

Since these studies, many complex mass storage systems have been implemented, including those at NASA Ames, NCAR, and SDSC [7,11,12]. However, no studies on these systems have been published. Instead, the data management staff at these sites collect huge amounts of data to plan for new equipment purchases and tune their systems. While this guarantees good performance for each system, it does not provide any guidelines for building future systems.

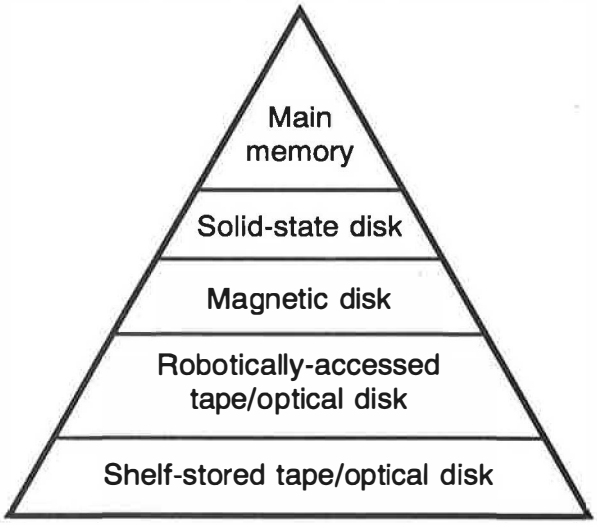


Figure 1: Memory and storage hierarchy in large computer systems. This is also called the "storage pyramid"

Mass Storage Devices

Currently, there are two major types of tertiary storage devices in common use – tape and optical disk. Both of these are high-density removable media. The tradeoffs between the two media are presented in Table 1. Two types of magnetic tape, helical scan and longitudinal (linear) scan, are presented. The numbers for the tapes come from [4], while the optical disk statistics come from [16]. Table 1 includes figures for the IBM 3490 and the Ampex D-2.

Currently, the IBM 3480 tape format is standard at most supercomputer installations, though some are beginning to move to helical scan tapes that provide higher density. The IBM 3480 uses linear recording, which provides high speed at the expense of recording density. The D-2 drive, on the other hand, uses helical scan techniques (similar to conventional VCR recording) to greatly increase recording density. With a new generation of linear tape being developed, however, both types of tape may be close in cost, performance and capacity.

The major tradeoffs among the three media are access latency and transfer bandwidth. Optical disks have a much lower access latency than either type of magnetic tape, but their bandwidth is also considerably lower. Thus, a system which performs many small I/Os to tertiary storage, such as a database system, would be best served by optical disk, since the dominating factor in calculating time per byte is access time to the first byte. For supercomputing installations, however, magnetic tape is better. While the time to get the first byte of data is longer for tape than for optical disk, the time to get *all* of the data is often lower for tape. Files on supercomputing installations tend to be large [20], so the difference in transfer time between optical disk and tape is substantial. In general, more expensive drives have higher transfer rate and storage density, though neither longitudinal scan nor helical scan seems intrinsically better. A new technology, optical tape [16], also looks promising because of its high density storage and high transfer rate.

Another primary consideration is price per gigabyte. As can be seen in Table 1, magnetic tape has a lower cost per gigabyte stored than optical disk. For systems with terabytes of data stored on tertiary storage, such as NCAR, this cost difference alone is enough to favor using tape exclusively as the tertiary store. The lower cost and higher transfer rate make magnetic tape the obvious choice for supercomputer centers which deal with sequentially-read large files.

Most installations today have one or more cartridge tape robots to automatically mount some of their tape libraries. The StorageTek 4400 [9] is an example of a tape robot, or automated cartridge system (ACS). This system can provide access to 1.2 TB of data (6000 IBM 3480 cartridges, holding 200

Category	Optical Disk Jukebox	Linear Tape	Helical-Scan tape
Media capacity (GB)	1.2	0.4	25
Random access speed	7 sec	13 sec	60+ sec
Transfer rate (MB/sec)	0.25	6.0	15
Media cost/GB	$80	$25	$2

Table 1: A brief comparison of optical disk and tape. The linear tape is an IBM 3490 (high-density version of the 3480), and the helical-scan tape is an Ampex D-2.

MB each). Loading a cartridge takes approximately 6 seconds; from there, tape characteristics are those of the IBM 3480 tape drives in the tape silo.

Previous Work

There have been several studies of actual file migration systems, but most are quite old and deal with different computing environments. We will summarize them here, and in a later section will compare the results of studying the current NCAR environment with the results of the earlier studies.

In [15] and [14], Smith studied the file system at the Stanford Linear Accelerator Center. His data dealt with Wylbur text editor data sets, and tracked the references to those data sets. He found that the best algorithms had access to the entire reference string for a file. Since this is often not feasible, the criterion he suggested was to migrate off disk the files with the highest value of

$$last\_reference\_time^{1.4} \times file\_size.$$

This algorithm, called Space-Time Product (STP**1.4), was the best of the algorithms examined which did not make use of any file history other than the last reference time. The analysis in the paper also did not consider the possible effects of transfer time and access latency in minimizing average file reference time; instead, the analysis attempted to minimize file miss rate.

Smith also made several observations about file system activity. He noted that usage followed a weekly pattern, with activity highest on weekdays and lower on weekends and holidays. He also has extensive data on file sizes and interreference intervals. Because of the size of the data set in our NCAR study (over 900,000 files), it would be very difficult to perform the same computations over the entire file set. The data set in Smith's paper has a granularity of one day and does not distinguish between reads and writes.

None of the acceptable migration algorithms would have had much effect on average file access time at NCAR. As noted in Smith's paper, a miss ratio of 1% would mean a loss of 6.26 person-minutes per day, given the file usage rates and the number of users on the system. For STP, this miss ratio would require a disk system that held 1.5% of the total tertiary storage, and would require 300 tracks, or about 1 MB, of data to be transferred each day.

Lawrie, et. al., in [10], considered the file migration patterns on the University of Illinois Cyber 175. Again, the system examined is quite different from the one studied in this paper. Interestingly, Lawrie reported that, though his system was quite different from SLAC, his results matched Smith's closely. This paper also examined several migration algorithms, and compared them against Smith's STP algorithm on their data. They found that STP was better than the algorithms they tried, which included pure LRU, pure length (migrate large files first), and SAAC, which migrated files that became less active. In all cases, STP outperformed these algorithms, though only by a slim margin.

Two recent studies focused on a workstation file system at Berkeley [17] and the Common File System (CFS) at the National Center for Supercomputing Applications (NCSA) [8]. At Berkeley, Strange found that there were more file reads than file writes, though more data was written than read. He also found that, as expected, less data was used on weekends (even though the system was primarily used by graduate students). In this system, algorithms using a space-time product to identify files to migrate would work well. However, files were much smaller than typical supercomputer files. Even the file system with the largest files averaged under 50 KB/file. As Table 4 shows, this is far smaller than typical supercomputer files. The file system profile at NCSA, on the other hand, is quite similar to that at NCAR. File sizes are similar, and file reference rates are close to those in this study. This gives us high confidence that NCAR is typical of supercomputer mass storage systems.

Other papers have simply presented data gathered from existing mass storage systems without analyzing the data and suggesting possible algorithm changes. Systems analyzed include Brookhaven [5], NCAR [1,18], and NASA [7]. In addition, many large sites internally publish a summary of statistics gathered from their machines. They use these statistics for two purposes: to better tune their systems, and to justify new equipment purchases.

3 NCAR system configuration

In this section, we describe the system on which we gathered the file migration traces. Rather than describe the entire NCAR network, we focus on the parts which are relevant to the study. However, the rest of the network will be briefly described, since the mass storage system is shared by all of the systems at NCAR, so their presence might affect mass storage systems performance.

Hardware Configuration

The CPU in the study was a Cray Y-MP 8/864 (`shavano.ucar.edu`), with 8 CPUs and 64 MWords of main memory. Each CPU has a 6 ns cycle time. `Shavano`, like other Cray Y-MPs, has several 100 MB/sec connections to its local disks and two 1 GB/sec connections to a solid state disk (SSD). There are about 56 GB of disks attached directly to the Cray; 47 GB of this space is reserved for application scratch space and files over a few days old are purged from it regularly.

The mass storage system (MSS) at NCAR is composed of an IBM 3090 – used as a bitfile server – with 100 GB of online disk on IBM 3380s, a StorageTek Automated Cartridge System 4400 with

6000 200 MB IBM 3480-style cartridges, and approximately 25 TB of data in shelved tape. The MSS tries to keep all files under 30 MB on the 3090 disks, and immediately sends all files over 30 MB to tape. Usually, the tapes written are those in the cartridge silo. Files on the MSS are limited to 200 MB in length, since a file cannot span multiple tapes. While the Cray supports much larger files on its local disks, they must be broken up before they can be written to the MSS.

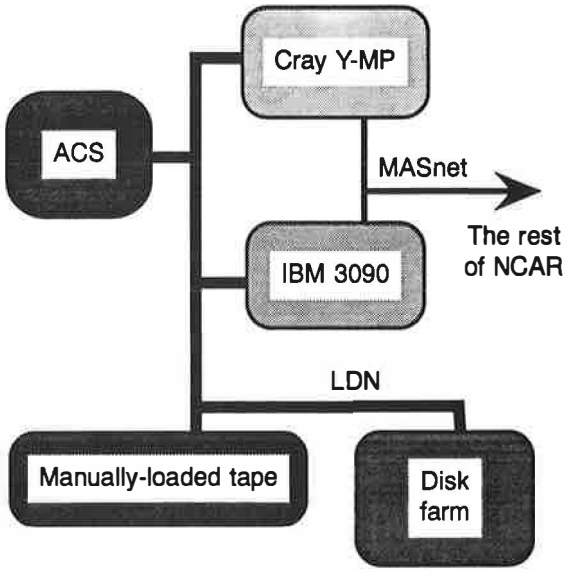


Figure 2: Network connections between machines at NCAR.

The MSS at NCAR is shared by the entire NCAR computing environment, which includes the Cray Y-MP, an IBM 3090 which runs the MSS, several VAXen, and many workstations. Figure 2 shows the network connections between the various machines at NCAR. The disks and tape drives attached to the MSS processor have direct connections to the Crays via the Local Data Network (LDN), providing a high-speed data path. All machines connected to the MSS (including the Crays) are connected to the 3090 by a custom hyperchannel-based network called the MASnet. Data going out over the MASnet must pass through the 3090's main memory, so it is a slower path than the direct connection the Crays have. The few workstations with connections act as gateways to the networks which connect to the rest of the workstations at NCAR. These gateways are also the fileservers for the local networks. Many of these smaller machines have their own local lower-speed disks, about 5.5 GB of which are mounted by the Cray via NFS (Network File System). According to the monthly report published by the NCAR systems group [20], `shavano` puts more data on the network than any other node, but several other nodes receive more data. In particular, several of the Sun workstations receive comparable amounts of data. It is likely that these workstations, which are the gateways to internal networks of desktop workstations, are receiving a large amount of image traffic.

System Software

The Cray Y-MP is primarily used for climate simulations – both the extensive number crunching necessary to generate the data, and the less computationally-intensive processing used in visualizing it. The Cray has two primary modes of operation; it can either run in primarily interactive mode, where programs are short and run as the user requests them, or in batch mode, where jobs are queued up and run when space and CPU time are available. There is no explicit switch between operating modes, but short interactive jobs typically have higher priority. There is less CPU time for running batch jobs during the day, because scientists are looking at results from previous batch jobs. At night, however, the CPU is mainly used to run large jobs requiring hours of CPU time. The MSS request patterns reflect these two different uses of the CPU, as will be shown below.

The software which runs the MSS is based on concepts in the Mass Storage Systems Reference Model [3]. It consists of software on the mass storage control processor (MSCP), which is the IBM 3090, and one or more bitfile mover processes on the Cray. Users on the Cray make explicit requests (via the UNICOS commands `lread` and `lwrite`) to read or write the MSS. These commands send messages to the MSCP, which locates the file and arranges for any necessary media mounts. The MSCP then configures the devices to transfer directly to the Cray. For disk and tape silo requests, these mounts are handled without operator intervention, but an operator must intervene to mount any non-silo tapes which are requested. After the data is ready to be transferred, the MSCP sends a message to a bitfile mover, which manages the actual data movement. When transfer is complete, the bitfile mover returns a completion status to the user.

Applications

The Cray at NCAR runs two types of jobs – interactive jobs, which finish quickly and require a short turnaround time, and batch jobs, which may require hours of CPU time but have no specific response time requirements.

A typical climate simulation, such as the Community Climate Model [21], might take produce many megabytes of data per hour, all of which would be stored on a tertiary store. This is an example of a batch job, since a researcher would submit the job and allow it to run overnight or longer. These jobs use a large amount of temporary disk storage as well as CPU time. The Y-MP at NCAR is configured with small, 300 MB user partitions. Each user is allocated a few megabytes on one partition,

which would be insufficient for storing the output of even one run of a climate model. Thus, the initial input to a climate model must come from the MSS, and any results must go back to the MSS. If the results are needed later, they must be retrieved from the MSS.

Interactive jobs, such as a "movie" of the results of a climate simulation, have much more stringent response time requirements. Typically, a user will initiate a command and expect a response quickly. According to [19], an interactive request must be satisfied in just a few seconds, or interactive behavior is lost. Nevertheless, the average response time to satisfy MSS requests is over 60 seconds; possible solutions to this problem will be discussed later.

4 Tracing Methods

4.1 Trace Collection

The data used in this study was gathered from system logs generated by the mass storage controller process and the bitfile mover processes. Approximately 50 MB of data was written to these logs per month. The system managers at NCAR use the data to plan future equipment acquisitions and improve performance on the current system. The logs also serve as proof that a requested transaction took place. The system managers occasionally use them to refute users who claim their files were written to the MSS and then disappeared.

The system log, as written by the mass storage management processes, contains a wealth of information. Much of it is either redundant or unnecessary for migration tracing. Information such as project number and user name are not needed for migration studies, since the user identifier is also reported. The traces are designed to be easily human-readable, so fields are always identified and dates and times are in human-readable form. In addition, each MSS request is assigned a sequence number, since there are several records in the system log which correspond to the same I/O. This is useful for assembling a single record for a migration trace. By processing the traces to remove redundant information and transforming the rest of the information into a form more easily machine-readable, the traces were cut from 50 MB per month to 10-11 MB per month. They could not be reduced further because file names are long and could not be compressed without losing information.

Trace Format

Once the system logs were copied to a local host, they were processed into a trace in a format that is easy for a trace simulator or analysis program to read. The traces were kept in ASCII text so they would be easy to read on different machines with different byte orderings. A list of the fields in the trace is in Table 2.

Very little information is common between two consecutive records except temporal information. Even so, the traces are compressed by recording times as differences from some previous time, as suggested in [13]. The start time for a MSS request is recorded as the elapsed time since the start time of the previous request, while the latency until the first byte is transferred (the startup latency) and the transfer time are recorded as durations. Start time and startup latency are measured in seconds, while transfer time is measured in milliseconds.

These were the precisions available from the original system logs. The only other commonality between consecutive requests might be the requesting user, so there is a bit in the flag field which indicates that the request was made by the same user who made the previous request. Directories, too, might be common between consecutive requests, but they would be harder to match. Future versions of the trace format may allow for full or partial paths to be obtained from previous records.

Field	Meaning
source	Device the data came from
destination	Device the data is going to
flags	Read/write, error information, compression information
start time	time in seconds since the previous start time
startup latency	time in seconds to start the transfer
transfer time	time in milliseconds to transfer the data
file size	file size in bytes
MSS file name	file name on the MSS
local file name	file name on the computer
user ID	user who made the request

Table 2: Information in a single trace record

5 Observations

The traces for this study were collected over a period of 24 months, from October, 1990 through September, 1992. Traces were available from the time the MSS came on-line in June, 1990, but the MSS was very lightly used for the first few months. We decided to omit this data and study the "steady-state" system.

Trace Statistics

Overall statistics for the trace period are shown in Table 3. The traces actually include 3,688,817 references, but 175,633 (4.76%) had errors. The most common error was the non-existence of a requested file. In such cases, it was impossible to include the reference in our analysis, since the file never existed and wasn't stored on any device. It might have been

possible to include references that encountered other errors, such as media errors and premature termination, but there were few enough that they would not affect the results.

Number of files	902772
Average file size	25 MB
Number of directories	143245
Largest directory	24926 files
Maximum directory depth	12
Total data in MSS	23 TB

Table 4: Statistics for a file store needed to satisfy all of the traced accesses

Table 4 contains data about the massive store that accesses were made to. This table only includes files which were referenced during the trace period. Since we had no data on the actual contents of the MSS, we assumed that only files actually referenced during the trace period existed on the mass store. This is a valid simplification, as there are only three kinds of files that are never explicitly read or written – large temporary files used by Cray applications, small files that fit into the 1 MB allocated for a each user's home directory, and system files such as binaries. The first category, temporary files, would be actively used for their entire lifetime, and discarded when no longer in use, never providing a chance to move them to long-term storage. Small user files, such as `.cshrc`, would never be migrated since they would be used too often. Even if they were migrated, they would only add approximately 4 GB of space to the MSS, assuming each of the 4,000 users filled their entire permanent allocation. System files, likewise, would probably be used often enough so they would not be evicted from disk. Additionally, most system files are read-only, eliminating the need to write any data to the MSS.

Latency to first byte

Figure 3 shows the total latency from when a request is made to the MSS until the data transfer actually starts. This time is composed of several elements – queueing time on the Cray, queueing time on the MSS, media mounting time, and seek time.

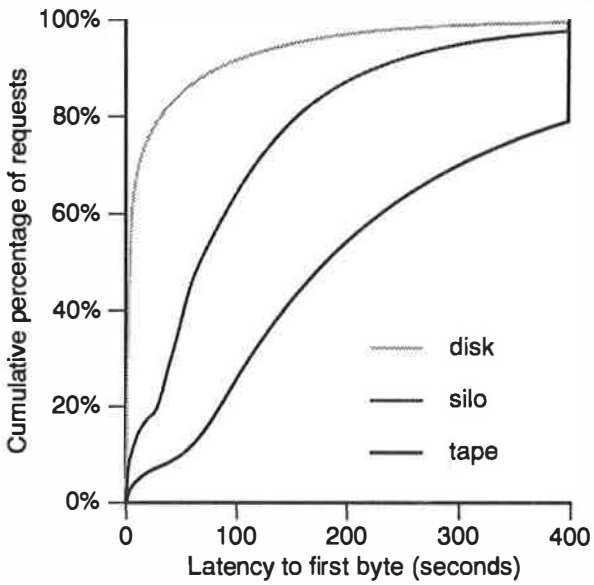


Figure 3: Latency to the first byte for various MSS devices

	Reads	Writes	Total
References	2336747 (66%)	1179047 (33%)	3515794 (100%)
Disk	1419280 (60%)	927722 (39%)	2347002 (66%)
Tape (silo)	480545 (66%)	239162 (33%)	719707 (20%)
Tape (manual)	436922 (97%)	12163 (2%)	449085 (12%)
GB transferred	63926.2 (73%)	23389.9 (27%)	87316.2 (100%)
Disk	5080.4 (58%)	3727.9 (42%)	8808.3 (10%)
Tape (silo)	38256.6 (67%)	19081.4 (33%)	57338.1 (66%)
Tape (manual)	20589.2 (97%)	580.6 (3%)	21169.8 (24%)
Avg. file size (MB)	27.36	19.84	24.84
Disk	3.58	4.02	3.75
Tape (silo)	79.61	79.78	79.67
Tape (manual)	47.12	47.74	47.14
Secs to first byte	98.1	38.6	78.18
Disk	32.47	25.39	29.67
Tape (silo)	115.14	81.86	104.08
Tape (manual)	292.58	203.84	290.18

Table 3: Overall trace statistics. The trace covers the period from October, 1990 through September, 1992. The percentages listed under "Reads" and "Writes" are ratios to the value in the "Total" column of that row. The percentages listed under "Total" are percentages relative to the top value in the column.

For the disk, media mounting time and seek time are very short, usually well under a second. While median access time for the disk was 4 seconds, the distribution has a long tail due to queueing at individual disks. Each disk has a relatively low bandwidth, so a large file takes several seconds to satisfy. Any requests for this disk that arrive in the meantime must wait for the long request to finish, generating the long delays in the tail of the disk latency curve.

Delays were caused by queueing in several places in the system – the Cray, the MSS CPU, the network from disk to Cray, and data transfer to or from the disk itself. Of these, the only delays that differ between disk, tape silo, and shelved tape are the latencies due to the device itself – transfer delays and seek delays. The disks do not transfer data much faster than the tape drives, so queueing delays for them are probably representative of the time spent waiting for data to be transferred off tape. We can then deduce how much extra time is needed by the tape systems to get the first byte of data.

The first observation is that the tape silo is considerably faster than manually fetching the tape. After subtracting off the queueing time exhibited by the disk, the silo is approximately 2 to 2.5 times as fast as the manual tape drives at getting to the first byte. Since the tape silo tape drives are the same as the operator-loaded tape drives, this difference must come from the time to mount the tape rather than from seek time. The StorageTek 4400 ACS can pick and mount a tape in under 10 seconds; after subtracting off average queueing time for the disk, which is 25 seconds, the non-seek overhead for reading an automatically-loaded tape is 35 seconds. According to Table 3, tape accesses take 85 seconds on average, so the average seek is 50 seconds long. When the same analysis is applied to manually loaded tapes, the manual tape mounting time is found to be approximately 115 seconds, or about 2 minutes. This is quite good. However, as Figure 3 shows, 10% of all manual tape mounts were not completed within 400 seconds. Nearly all of the tape silo and disk requests were completed by this time. This is probably the biggest weakness of manual tape mounting-the very long tail of the mounting time distribution. While other data accesses will almost certainly complete in 5 minutes, manual tape mounts may take much longer.

This is just a simple analysis, though. There are several factors that we did not consider which may affect our conclusions here. In particular, queueing time for the tape silo may be different from queueing time for the disks. There are only a few tape robots in the silo, and each is tied up for several seconds with a tape load. If several tape loads come in close together, some of them will have relatively long queueing times. This does not happen with disk, as each disk is tied up for relatively little time with each request.

Another observation is the relation between latency to access the first byte and time required for the entire transfer. Both the tapes and the disks can transfer at a peak rate of 3 MB/sec, but the observed rates are usually closer to 2 MB/sec. As a result, the transfer times are similar for the two media. For tape, an average file of 80 MB will take 40 seconds to transfer. This is comparable to the additional 60 second overhead from using tape instead of disk. One possible way to improve perceived response time in the system would be to use cut-through, as in [7]. Under this scheme, a call to open a file returns immediately, while the operating system continues to load the file from the MSS and keep track of how far it has gotten. When future requests are made, the call returns immediately unless the requested data has not yet been read. This scheme works because applications often do not read data as fast as the MSS can deliver it. Instead of delaying the application, then, it allows the application and file retrieval from the MSS to overlap. This system would be difficult to use in the current NCAR configuration, however, since the MSS is not seamlessly integrated with the local disk file system. The bitfile mover processes would have to have special communication protocols with the local file system to let it know how much of the file has been transferred. Nevertheless, it is a useful optimization and should be considered.

5.2 MSS usage patterns

Figure 4 shows the average amount of data transferred each hour of the day. As expected, activity is highest during working hours – from 9 AM to 5 PM. The variation in transfer rate, how-

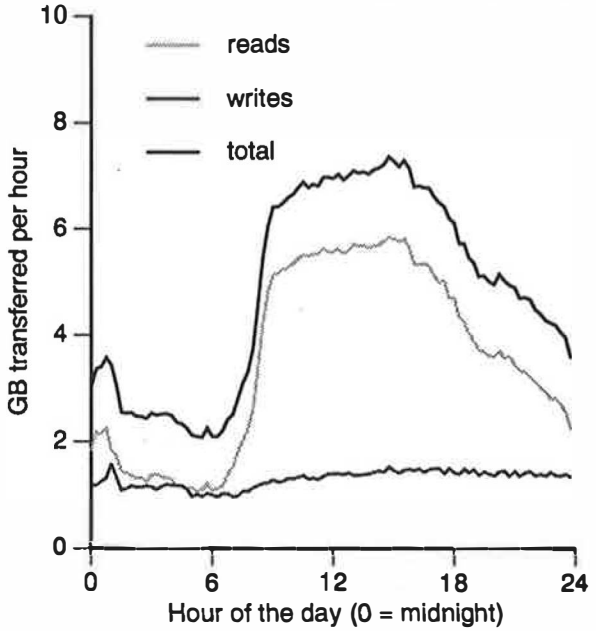


Figure 4: Average data transfer rate over the course of a day.

ever, is almost entirely due to reads. The amount of data read jumps greatly at 8 AM when the scientists usually arrive, and slowly tails off after 4 PM as they leave. The fall is slower than the rise because most scientists are more likely to stay late than to arrive early. This suggests that most reads on the system are initiated by interactive requests, since reads peak when people are at work, while writes remain almost constant regardless of the number of humans requesting data. File request rate over the course of a day shows a pattern similar to that of data transfer rate.

The weekly data transfer rates, shown in Figure 5, have patterns similar to those in the daily averages. As expected, read activity is lower on the weekends, since there are fewer researchers around to initiate read requests. Write requests, on the other hand, experience little variation over the course of the week, as the Cray CPU runs batch jobs all weekend. There is a small increase in write requests during the day, indicating that users do actually make some write requests; however, the change is small relative to the flood of read requests that users generate.

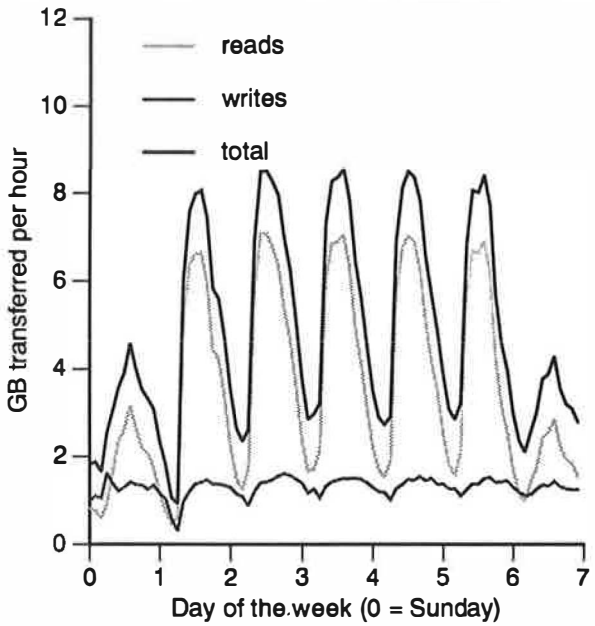


Figure 5: Average data transfer rate over the course of a week

Note that less data is transferred early Monday morning than on any other day. This low point can be attributed to two factors. First, the Cray might be taken down early on Monday morning for maintenance, as that would cause the least disruption of normal work. Second, any idle time the Cray might have would be on Monday morning, as the queues from the weekend might have finished.

Over the two years the trace covers, the mass storage system received increasingly large amounts of work. The average data rate for each of the 104 weeks is shown in Figure 6. There are drops in read request rate around Thanksgiving and Christmas for both 1990 and 1991. Note, however, that write request rate does not drop on these holidays. In fact, write requests *incrased* at the end of the year. This reinforces our conclusion that reads are interactive while writes are requested primarily by batch jobs, as the Cray doesn't take a Christmas vacation while the scientists do.

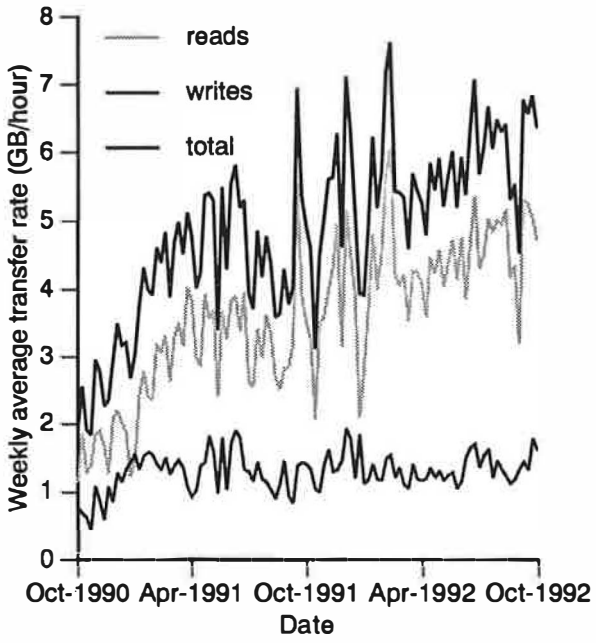


Figure 6: Average data transfer rate over the course of a week

The MSS data request rate increases over the period shown by the graph, but this gain is due almost entirely to increases in read requests. MSS write rate appears to be related to the speed with which the computer can generate results, while read rate is set by the number of users that want to read their data back. The lack of increase in write rate suggests that the Cray is already running at full capacity, and that researchers are simply using the machine more for tasks such as visualization of the results. A faster machine would then need a higher write rate to massive storage. There would be at least a corresponding increase in read rate, and it might be greater if the user community gets larger.

Interreference intervals

Figure 7 shows the distribution of intervals between references to the MSS. Since about 3,500,000 files were referenced over a period of 731 days (approximately 6.3×10^7 seconds), the average interval between MSS requests was 18 seconds.

Looking at the graph, however, shows that 90% of all references followed another by less than 10 seconds. This distribution suggests that I/Os are

clustered. There are several possible explanations for this. Since Cray files can be of (nearly) unlimited length, but files on the MSS cannot exceed 200 MB, clustering could occur since several files are accessed together by the same program. Another possibility is that there are really two distributions for intervals – those made by researchers' interactive requests, and those made by batch jobs. The interactive requests are very likely to be bunched together, since a researcher interested in day 1 of a climate model simulation will usually be interested in day 2, and both days will probably be in separate files.

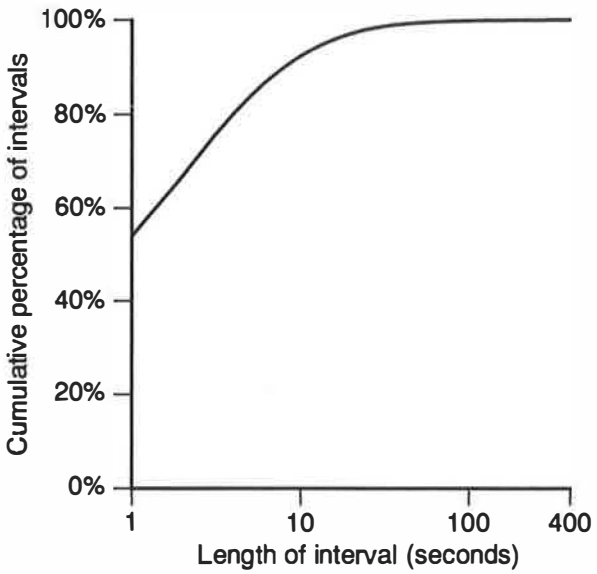


Figure 7: Lengths of intervals between Cray references to the MSS.

File reference patterns

Instead of counting all file references, this part of the analysis included at most one read and one write from any eight hour period. Since files on the MSS were explicitly referenced Unix command, some files were accessed many times in a short time. In a system with automatic migration, this would not be likely to happen.

As expected, most files were not referenced often. Figure 8 shows that only 5% of all files are referenced more than ten times. 50% of the files in the trace were never read at all, and another 25% were read only once. Writes were slightly different – just over 20% of the files were not written during the trace period, but another 65% were written exactly once. Of course, these numbers add up to more than 100%, as many files were read and written one time or less. In all, 57% of the files were accessed exactly once, and 19% were accessed exactly twice. Thus, only a quarter of the files were accessed more than two times. Our observations found that the median number of file references was one, as opposed to [14], which reported the median to be two. Furthermore, fully 44% of all the files in the trace were written exactly once and never read. These numbers confirm the common belief that many files are written to a massive store once and never read again.

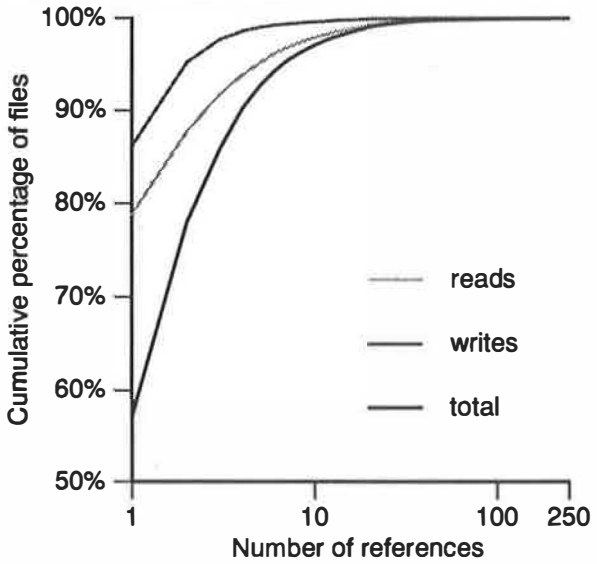


Figure 8: Distribution of file reference counts. During the trace period, 50% of the files had 0 reads and 21% had 0 writes.

Figure 9 shows the distribution of time intervals between references to a given file, called *interreference intervals*. Long interreference intervals mean that a file is referenced infrequently, while short intervals indicate many accesses over a short

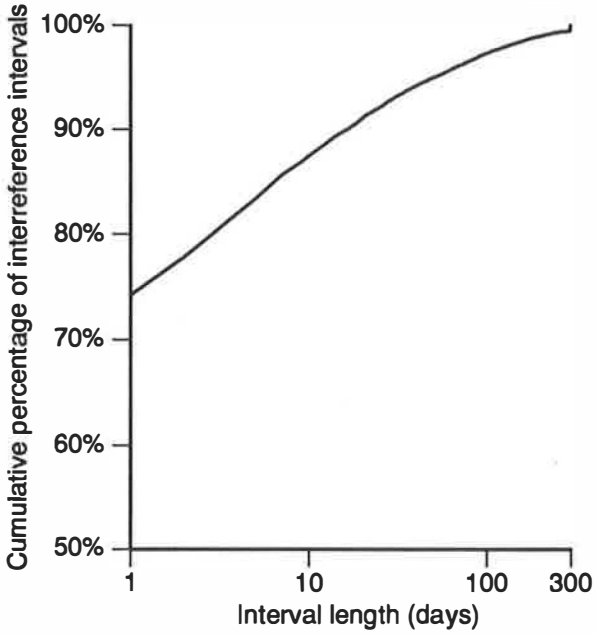


Figure 9: Distribution of intervals between successive references to the same file. 70% of all intervals were less than 1 day.

period of time. As Figure 9 shows, interreference intervals were short. This means that, for files which were rereferenced, the second access came soon after the first. Note, however, that there were still some files that were referenced more than a year after the previous reference to them. These references could not be easily predicted, so it is not sufficient merely to use prediction to improve access time; we must decrease the latency for random requests as well.

File and directory sizes

The dynamic distribution of file sizes transferred between the MSS and the Cray is shown in Figure 10. In this graph, a file is counted once for each access to it. The distributions of files read and files written are similar, though there is a small jump in file writes at approximately 8 MB. However, 40% of all requests are for files 1 MB or smaller. Since reads are more likely than writes to be initiated by a human user (as shown earlier), this graph suggests that performance on small file reads in a migration system would be especially important. Such small files make up under 1% of the total data storage requirement, so it seems wise to store these files on inexpensive, low-performance disks rather than on tape. If magnetic disk would be too expensive, an optical disk jukebox could provide low latency to the first byte and high capacity.

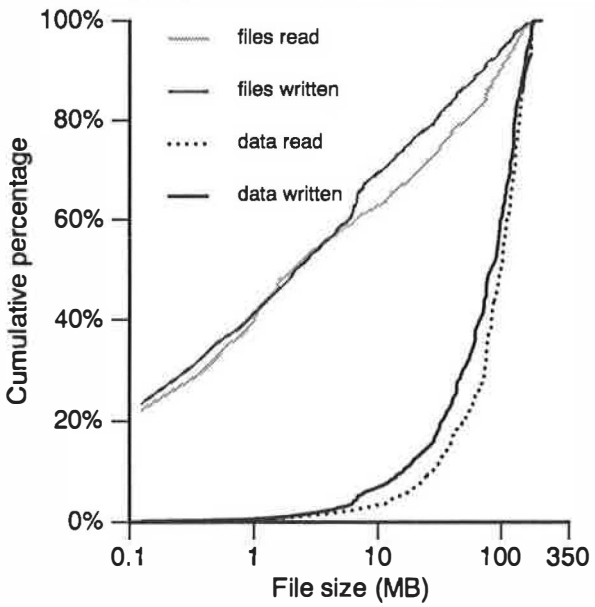


Figure 10: Size distribution of files transferred between the MSS and the Cray. A file is counted once for each time it is requested.

The distribution of file sizes on the MSS during the trace period is graphed in Figure 11. In it, each referenced file is counted exactly once, regardless of the number of times it was accessed. The graph shows that, while about half of the files are under 3 MB, these files contain 2% of the data. Algorithms that take file size as an argument could use this fact to simplify their bookkeeping, as all files below a threshold size could be considered equivalent when computing space-time products. Since most files are below this size, the algorithm should run much faster.

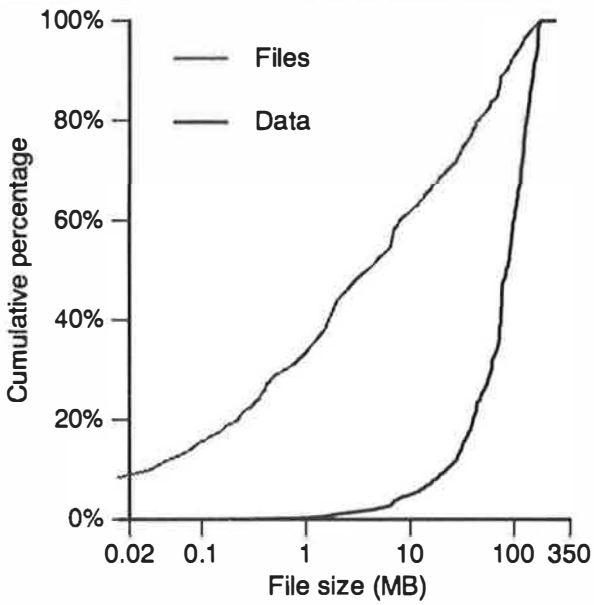


Figure 11: Distribution of file sizes on the MSS. Each file referenced in the trace is counted once.

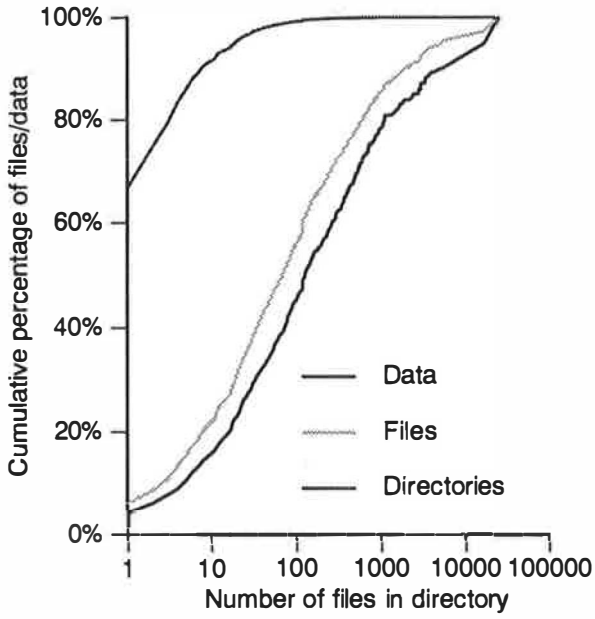


Figure 12: Distribution of directory sizes on the MSS. Note that more than half of the directories had only zero or one file in them (though most of these also had subdirectories). Note also that 5% of the directories held 50% of the files and data.

Directories also tended to be small. Figure 12 shows that 90% of the directories had 10 or fewer files, and 75% had only zero or one file. Even so, over half of all files and data were in large directories that contained more than 100 files. The size and number of directories is very important, as many current systems do not archive directories or file metadata such as inodes. With over 130,000 directories and 900,000 files, the NCAR system needs to store gigabytes of metadata on disk. Future systems must be able to move this information to tape, especially since over 40% of the metadata describes files that will never be accessed again.

6 File migration algorithms

The observations made from the NCAR trace data have several implications for future file migration algorithms. The system studied here is quite different from that in the studies done around 1980 [6,15]; while file access patterns are not radically different, the files themselves are larger and there are more of them.

The NCAR system uses two different migration algorithms – one for moving files between the Cray and the MSS, and the other for relocating files on different media within the MSS. Moving files between the Cray and the MSS is entirely manual, so there is no choice in the "algorithm" involved. However, using automatic migration between the Cray and the MSS would still save many file requests. About one third of all requests came within eight hours of another request for the same file. Often, these accesses are generated by batch job scripts which must read or write files on the MSS. If several of these scripts are run at about the same time, the Cray must make a separate request to the MSS for each script; it has no way of keeping track of multiple references to the same file. Better integration of the MSS with the Cray would fix this problem.

Another change since 1980 involves large files. Previous algorithms optimize for low seek time and ignore transfer time. For multi-megabyte files, transfer time dominates the time needed to access a file. On magnetic disk, seek time is far lower than transfer time for megabyte-sized files. Even for robotic tape, however, seek time is comparable to transfer time. A StorageTek robot can load a 3480 tape in under 10 seconds; the drive can transfer 20 MB in this time. The standard algorithms all make the assumption that the retrieval cost is the same for all files (though the storage cost may not be). New algorithms will have to take the difference in access time into account. The NCAR system already does this by storing smaller files on magnetic disk and larger files only on tape. In this way, small files do not suffer the latency penalties of tape. Large files, on the other hand, must wait for a tape to be loaded. However, their transfer time is long enough that the added delay of loading a tape is not as noticeable. The dividing point between storing files on disk and storing them on tape is a subject for future research; however, it is likely that the switchover point will be a function of tape seek speed and transfer rate.

Previous algorithms also make little distinction between reads and writes, primarily because their trace-gathering methods did not allow them to distinguish between such accesses. However, this difference is crucial for a file migration algorithm. The read/write ratio to the MSS at NCAR is 2:1, contrasting with conventional wisdom that an MSS serviced more writes than reads. Additionally, humans must wait for the results from reads, while users would not need to wait for writes to tape to complete. This suggests that an algorithm should not wait until it is absolutely necessary to free up space; instead, it should write data to tape relatively quickly, and then mark the file as "deleteable." Since files would be written lazily, their media placement could be optimized, thus speeding future reads. A mass storage system should be optimized to make read access to files faster at the cost of requiring more work for writes. This will make the system seem faster to its users at little additional cost.

7 Conclusions

This analysis of file movement between secondary and tertiary storage at a supercomputer Unix site provides several important hints for designers of file migration systems. First, humans wait for reads, while computers wait for writes. Any migration policy should consider this, and optimize for reading. The write rate is relatively steady over time, while reads vary greatly. Thus, migration algorithms should move files to tertiary storage whenever resources (tape drives, etc.) are available, and use the extra space to prefetch files which might be read shortly.

Files have become larger and more numerous since the early 1980s. Currently, there are over 900,000 files on the MSS at NCAR averaging over 25 MB each. On the other hand, their reference patterns have not changed much. File rereference rate still drops off sharply after the first few days, though it does level off soon thereafter. Files are also infrequently rereferenced; more than half of the files were only accessed once in two years. Again, this suggests that files can be migrated to a less costly storage medium if they are unreferenced for only a few days.

The NCAR system appears to be a typical large Unix-based scientific computing center. Thus, the analysis in this paper will help system architects design hardware and software best suited for storing and rapidly accessing the terabytes of data that such systems must store. While reference patterns for these data have not changed much in the last decade, more files, larger files and new tertiary storage

technologies will require new mass storage systems and new migration algorithms to run them.

Acknowledgments

The authors would like to thank the staff at NCAR for all their help in gathering the traces and understanding their format. Special thanks go to Bernie O'Lear for providing access to the data at NCAR and to Dennis Colarelli for his assistance with the NCAR system. We would also like to thank our colleagues at Berkeley and elsewhere for their helpful comments on drafts of this paper.

References

[1] Edward R. Arnold and Marc E. Nelson. "Automatic Unix backup in a mass-storage environment." In USENIX – Winter 1988, pages 131-136, February 1988.

[2] Donald L. Boyd. "Implementing mass storage facilities in operating systems." Computer, pages 40-45, February 1978.

[3] Sam Coleman and Steve Miller. "Mass storage system reference model: Version 4." IEEE Technical Committee on Mass Storage Systems and Technology, May 1990.

[4] Ann L. Drapeau and Randy H. Katz. "Striped tape arrays." In Digest of Papers. Twelfth IEEE Symposium on Mass Storage Systems, 1993. To appear.

[5] Carrel W. Ewing and Arnold M. Peskin. "The Masstor mass storage product at Brookhaven National Laboratory." Computer, pages 57-66, July 1982.

[6] Gordon George Free. "File migration in a UNIX environment." Master's thesis, University of Illinois at Urbana-Champaign, December 1984.

[7] Robert L. Henderson and Alan Poston. "MSS II and RASH: A mainframe UNIX based mass storage system with a rapid access storage hierarchy file management system." In USENIX – Winter 1989, pages 65-84, 1989.

[8] David W. Jensen and Daniel A. Reed. "File archive activity in a supercomputer environment." Technical Report UIUCDCS-R-91-1672, University of Illinois at Urbana-Champaign, April 1991.

[9] David D. Larson, James R. Young, Thomas J. Studebaker, and Cynthia L. Kraybill. "StorageTek 4400 automated cartridge system." In Digest of Papers, pages 112-117. Eighth IEEE Symposium on Mass Storage Systems, November 1987.

[10] Duncan H. Lawrie, J. M. Randal, and Richard R. Barton. "Experiments with automatic file migration." Computer, pages 45-55, July 1982.

[11] Fred W. McClain. "Mass storage at the San Diego Supercomputer Center." In Digest of Papers, pages 81-86. Eighth IEEE Symposium on Mass Storage Systems, November 1987.

[12] Marc Nelson, David L. Kitts, John H. Merrill, and Gene Harano. "The NCAR mass storage system." In Digest of Papers. Eighth IEEE Symposium on Mass Storage Systems, November 1987.

[13] A. Dain Samples. "Mache: No-loss trace compaction." Technical Report UCB/CSD 88/446, University of California at Berkeley, September 1988.

[14] Alan Jay Smith. "Analysis of long term file reference patterns for application to file migration algorithms." IEEE Transactions on Software Engineering, 7(4):403-417, July 1981.

[15] Alan Jay Smith. "Long term file migration: Development and evaluation of algorithms." Communications of the ACM, 24(8):521-532, August 1981.

[16] Ken Spencer. "Terabyte optical tape recorder." In Digest of Papers, pages 144-146. Ninth IEEE Symposium on Mass Storage Systems, November 1988.

[17] Stephen Strange. "Analysis of long-term UNIX file access patterns for application to automatic file migration strategies." Technical Report UCB/CSD 92/700, University of California, Berkeley, August 1992.

[18] Erich Thanhardt and Gene Harano. "File migration in the NCAR mass storage system." In Digest of Papers, pages 114-121. Ninth IEEE Symposium on Mass Storage Systems, November 1988.

[19] David Tweten. "Hiding mass storage under UNIX: NASA's MSS-II architecture." In Digest of Papers, pages 140-145. Tenth IEEE Symposium on Mass Storage Systems, May 1990.

[20] Sandra J. Walker. "Cray Computer, MSS, MASnet, MIGS and UNIX, Xerox 4050, 4381 Front-End, Internet Remote Job Entry, Text and Graphics System, March 1991." Technical report, National Center for Atmospheric Research, Scientific Computing Division, March 1991.

[21] David L. Williamson, Jeffrey T. Kiehl, V. Ramanathan, Robert E. Dickinson, and James J. Hack. "Description of NCAR Community Climate Model (CCM1)." Technical Report NCAR/TN-285+STR, National Center for Atmospheric Research, June 1987.

Author Information

Ethan Miller received a BS in computer science from Brown in 1987, and an MS from Berkeley in 1990. He is currently a PhD candidate in computer science at Berkeley, where he is a member of the RAID project. He is interested in file systems and data storage for high performance computing,

including both disk and tertiary storage. His U. S. Mail address is Computer Science Division; 571 Evans Hall; University of California; Berkeley, CA 94720. Electronic mail sent to elm@cs.Berkeley.EDU will also get to him.

Randy Katz has been on the Berkeley faculty since 1983. He received his MS and PhD at Berkeley in 1978 and 1980 respectively. He received his AB degree from Cornell University in 1976. He is the principal investigator of a DARPA and NASA sponsored project to construct high performance, high capacity storage systems for diskless supercomputers. His U. S. Mail address is the same as above, and his e-mail address is randy@cs.berkeley.edu.

HighLight: Using a Log-structured File System for Tertiary Storage Management[1]

John T. Kohl – University of California, Berkeley and Digital Equipment Corporation
Carl Staelin – Hewlett-Packard Laboratories
Michael Stonebraker – University of California, Berkeley

ABSTRACT

Robotic storage devices offer huge storage capacity at a low cost per byte, but with large access times. Integrating these devices into the storage hierarchy presents a challenge to file system designers. Log-structured file systems (LFSs) were developed to reduce latencies involved in accessing disk devices, but their sequential write patterns match well with tertiary storage characteristics. Unfortunately, existing versions only manage memory caches and disks, and do not support a broader storage hierarchy.

HighLight extends 4.4BSD LFS to incorporate both secondary storage devices (disks) and tertiary storage devices (such as robotic tape jukeboxes), providing a hierarchy within the file system that does not require any application support. This paper presents the design of HighLight, proposes various policies for automatic migration of file data between the hierarchy levels, and presents initial migration mechanism performance figures.

Introduction

HighLight combines both conventional disk secondary storage and robotic tertiary storage into a single file system. It builds upon the 4.4BSD LFS [10], which derives directly from the Sprite Log-structured File System (LFS) [9], developed at the University of California at Berkeley by Mendel Rosenblum and John Ousterhout as part of the Sprite operating system. LFS is optimized for writing data, whereas most file systems (e.g., the BSD Fast File System [4]) are optimized for reading data. LFS divides the disk into 512KB or 1MB segments, and writes data sequentially within each segment. The segments are threaded together to form a log, so recovery is quick; it entails a roll-forward of the log from the last checkpoint. Disk space is reclaimed by copying valid data from dirty segments to the tail of the log and marking the emptied segments as clean.

Since log-structured file systems are optimized for write performance, they are a good match for the write-dominated environment of archival storage. However, system performance will depend on optimizing read performance, since LFS already optimizes write performance. Therefore, migration policies and mechanisms should arrange the data on tertiary storage to improve read performance.

HighLight was developed to provide a data storage file system for use by Sequoia researchers. Project Sequoia 2000 [14] is a collaborative project between computer scientists and earth science researchers to develop the necessary support structure to enable global change research on a larger scale than current systems can support. HighLight is one of several file management avenues under exploration as a supporting technology for this research. Other storage management efforts include the Inversion support in the POSTGRES database system [7] and the Jaquith manual archive system [6] (which was developed for other uses, but is under consideration for Sequoia's use).

The bulk of the on-line storage for Sequoia will be provided by a 600-cartridge Metrum robotic tape unit; each cartridge has a capacity of 14.5 gigabytes for a total of nearly 9 terabytes. We also expect to have a collection of smaller robotic tertiary devices (such as the Hewlett-Packard 6300 magneto-optic changer). HighLight will have exclusive rights to some portion of the tertiary storage space.

HighLight is currently running in our laboratory, with a simple automated file-oriented migration policy as well as a manual migration tool. HighLight can migrate files to tertiary storage and automatically fetch them again from tertiary storage into the cache to enable application access.

The remainder of this paper presents HighLight's mechanisms and some preliminary performance measurements, and speculates on some useful migration policies. We begin with a thumb-

[1]This research was sponsored in part by the University of California and Digital Equipment Corporation under Digital's flagship research project "Sequoia 2000: Large Capacity Object Servers to Support Global Change Research." Other industrial and government partners include the California Department of Water Resources, United States Geological Survey, MCI, ESL, Hewlett Packard, RSI, SAIC, PictureTel, Metrum Information Storage, and Hughes Aircraft Corporation. This work was also supported in part by Digital Equipment Corporation's Graduate Engineering Education Program.

nail sketch of the basic Log-structured file system, followed by a discussion of our basic storage and migration model and a comparison with existing related work in policy and mechanism design. We continue with a brief discussion of potential migration policies and a description of HighLight's architecture. We present some preliminary measurements of our system performance, and conclude with a summary and directions for future work.

LFS Primer

The primary characteristic of LFS is that all data are stored in a segmented log. The storage consists of large contiguous spaces called *segments* which may be threaded together to form a linear log. New data are appended to the log, and periodically the system checkpoints the state of the system. During recovery the system will roll-forward from the last checkpoint, using the information in the log to recover the state of the file system at failure. Obviously, as data are deleted or replaced, the log contains blocks of invalid or obsolete data, and the system must coalesce this wasted space to generate new, empty segments for the log.

4.4BSD LFS shares much of its implementation with the Berkeley Fast File System (FFS) [4]. It has two auxiliary data structures not found in FFS: the *segment summary table* and the *inode map*. The segment summary table contains information describing the state of each segment in the file system. Some of this information is necessary for correct operation of the file system, such as whether the segment is clean or dirty, while other information is used to improve the performance of the cleaner, such as the number of live data bytes in the segment. The inode map contains the current disk address of each file's inode, as well as some auxiliary information used for file system bookkeeping. In 4.4BSD LFS, both the inode map and the segment summary table are contained in a regular file, called the *ifile*.

When reading files, the only difference between LFS and FFS is that the inode's location is variable. Once the system has found the inode (by indexing the inode map), LFS reads occur in the same fashion as FFS reads, by following direct and indirect block pointers[2]. When writing, LFS and FFS differ substantially. In FFS, each logical block within a file is assigned a location upon allocation, and each subsequent operation (read or write) is directed to that location. In LFS, data are written to the tail of the log each time they are modified, so their location changes. This requires that their index structures (indirect blocks, inodes, inode map entries, etc.) be updated to reflect their new location, so these index structures are also appended to the log.

[2]LFS and FFS share this indirection code in 4.4BSD.

In order to provide the system with a ready supply of empty segments for the log, a user-level process called the *cleaner* garbage collects free space from dirty segments. The cleaner selects one or more dirty segments to be cleaned, appends all valid data from those segments to the tail of the log, and then marks those segments clean. The cleaner communicates with the file system by reading the ifile and calling a handful of LFS-specific system calls. Making the cleaner a user-level process simplifies the adjustment of cleaning policies.

For recovery purposes the file system takes periodic checkpoints. During a checkpoint the address of the most recent *ifile* inode is stored in the superblock so that the recovery agent may find it. During recovery the threaded log is used to roll forward from the last checkpoint. Each segment of the log may contain several *partial segments*. A partial segment is considered an atomic update to the log, and is headed by a segment summary cataloging its contents. The summary also includes a checksum to verify that the entire partial segment is intact on disk and provide an assurance of atomicity. During recovery, the system scans the log, examining each partial segment in sequence.

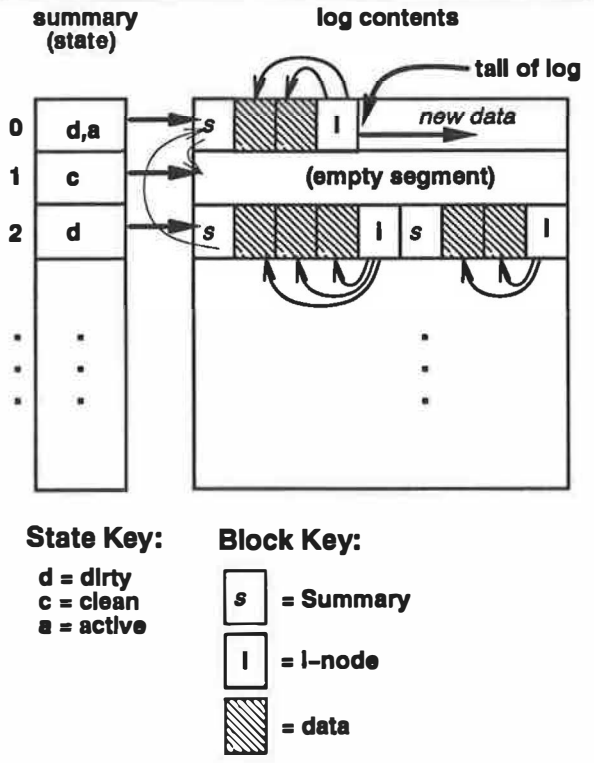


Figure 1: LFS data layout

Figure 1 shows the on-disk data structures of 4.4BSD LFS. The on-disk data space is divided into segments. Each segment has a summary of its state (whether it is clean, dirty, or active). Dirty segments contain live data (data which are still accessible to a user, i.e., not yet deleted or replaced). At

the start of each segment there is a summary block describing the data contained within the segment and pointing to the next segment in the threaded log. In Figure 1 we have shown three segments, numbered 0, 1, and 2. Segment 0 contains the current tail of the log. New data is being written to this segment, so it is both active and dirty. Once Segment 0 fills up the system will begin writing to Segment 1, which is currently clean and empty. Segment 2 was written just before Segment 0; it is dirty and contains live data.

Storage and Migration Model

HighLight has a "disk farm" to provide rapid access to file data, and one or more tertiary storage devices to provide vast storage. It manages the storage and the migration between the two levels. The basic storage and migration model is illustrated in Figure 2. HighLight has a great deal of flexibility, allowing arbitrary data blocks, directories, indirect blocks, and inodes to migrate to tertiary storage at any time. It uses the basic LFS layout to manage the on-disk storage and applying a variant on the cleaning mechanism to provide the migration mechanism. A natural consequence of this layout is the use of LFS segments for the tertiary-resident data representation. By migrating segments, it is possible to migrate some data blocks of a file while allowing others to remain on disk if a file's blocks span more than one segment.

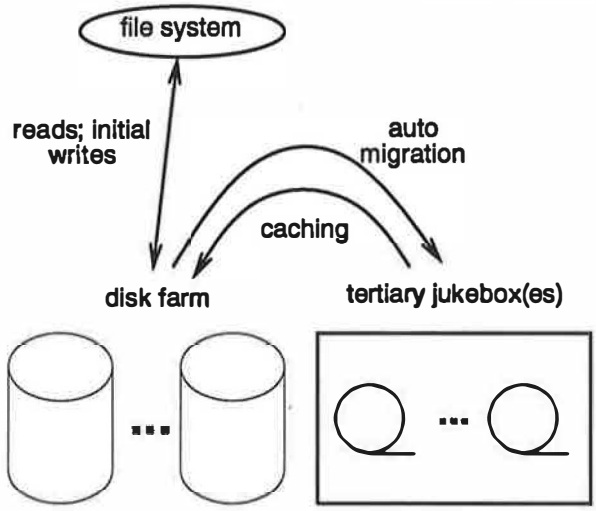


Figure 2: The storage hierarchy

Data begin life on the "disk farm" when they are created. A file (or part of it) may eventually migrate to tertiary storage according to a migration policy. The to-be-migrated data are moved to an LFS segment in a staging area, using a mechanism much like the cleaner's normal segment reclamation. When a staging segment is filled, it is written to tertiary storage as a unit.

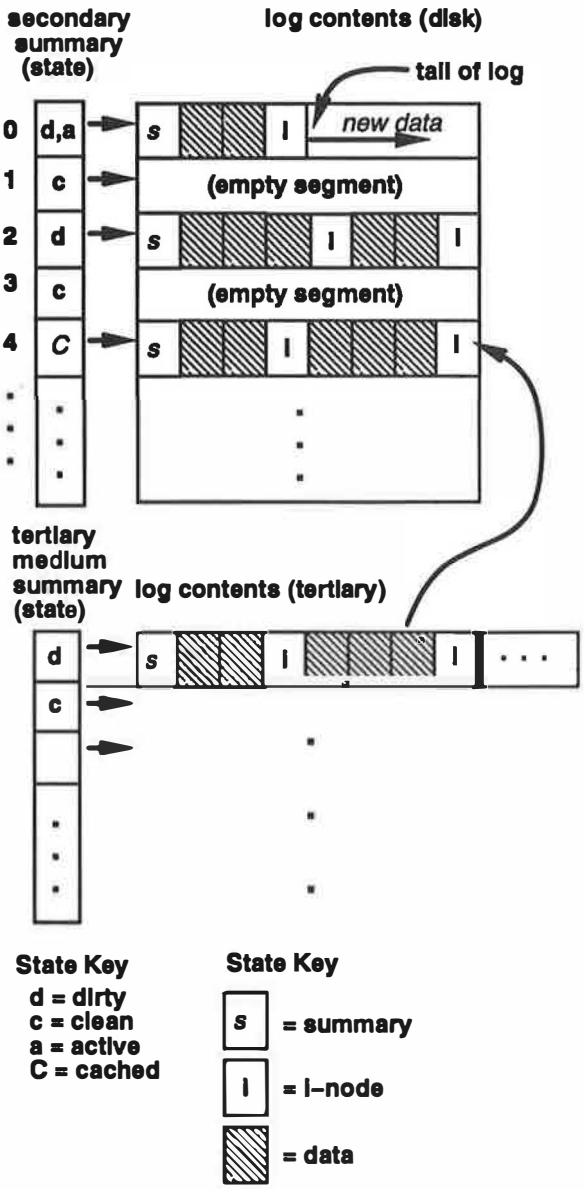


Figure 3: HighLight data layout

When tertiary-resident data are referenced, their containing segment(s) are fetched into the disk cache. These read-only cached segments share the disk with active non-tertiary segments. Figure 3 shows a sample tertiary-resident segment cached in a disk segment. Data in cached tertiary-resident segments are not modified in place on disk; rather, any changes are appended to the LFS log in the normal fashion. Since cached segments never contain the sole copy of a block, they may be flushed from the cache at any time if the space is needed for other cache segments or for new data.

Related Work

Some previous studies have considered automatic migration mechanisms and policies for tertiary storage management. Strange [16] develops a

migration model based on daily "clean up" computation which migrates candidate files to tertiary storage once a day, based on the next day's projected need for consumable secondary storage space. While Strange provides some insight on possible policies, we prefer not to require a large periodic migration run (our eventual user base will likely span many time zones, so there may not be any good "dead time" during which to process migration needs); instead we require the ability to run in continuous operation.

Unlike whole-file migration schemes such as Strange's or UniTree's [2], we want to allow migration of portions of files rather than whole files. Our partial-file migration mechanism can support whole file migration, if desired for a particular policy. We also desire to allow file system metadata, such as directories, inode blocks or indirect pointer blocks, to migrate to tertiary storage[3]. A final reason why existing systems may not be applicable to Sequoia's needs lies with the expected access patterns. Smith [11 ,12] studied file references based mostly on editing tasks; Strange [16] studied a networked workstation environment used for software development in a university environment. Unfortunately, those results may not be directly applicable for our environment, since we expect Sequoia's file system references to be generated by database, simulation, image processing, visualization, and other I/O intensive-processes [14]. In particular, the database reference patterns will be query-dependent, and will most likely be random accesses within a file rather than sequential access.

Our migration scheme is most similar to that described by Quinlan [8] for the Plan 9 file system. He provides a disk cache as a front for a WORM device which stores all permanent data. When file data are created, their tertiary addresses are assigned but the data are only written to the cache; a nightly conversion process copies that day's fresh blocks to the WORM device. A byproduct of this operation is the ability to "time travel" to a snapshot of the filesystem at the time of each nightly conversion. Unlike that implementation, however, we do not wish to be tied to a single tertiary device and its characteristics (we may wish to reclaim tertiary storage), nor do we provide time travel. Instead we generalize the 4.4BSD LFS structure to enable migration to and from any tertiary device with sufficient capacity and features.

[3]A back-of-the-envelope calculation suggested by Ethan Miller shows why: Assuming 200MB files and a 4K block size, we have an overhead of about 0.1% (200K) for indirect pointer blocks using the FFS indirection scheme. A 10TB storage area then requires 10GB of indirect block storage. Why not use this 10GB for cache area instead of wasting it on indirect blocks of files that lay fallow?

The key combination of features which we provide are: the ability to migrate all file system data (not just file contents); tertiary placement decisions made at migration time, not file creation time; data migration in units of LFS segments; migration performed by user-level processes; and migration policy implemented by a user-level process.

Migration Policies

Because HighLight includes a storage hierarchy, it must move data up and down the hierarchy. Migration policies may be considered in two parts, writing to tertiary storage, and caching from tertiary storage.

Before describing our migration policies, we must first state our initial assumptions regarding file access patterns, which are based on previous analyses of systems [5, 16, 11]. Our basic assumptions are that file access patterns are skewed, and that most archived data are never re-read. However, some archived data will be accessed, and once archived data became active again, they will be accessed many times before becoming inactive again.

Since HighLight optimizes writes by virtue of its logging mechanism, migration policies must be aimed at improving read performance. When data resident on tertiary storage is cached on secondary storage and read, the migration policy should have optimized the layout so that these read operations are as inexpensive as possible. There needs to be a tight coupling between the cache fetch and migration policies.

HighLight has one primary tool for optimizing tertiary read performance: segment reads. When data are read from tertiary storage, a whole 1MB segment (which is the equivalent of a cache line in processor caches) is fetched and placed in the segment cache, so that additional accesses to data within the segment proceed at disk access rates. Policies used with HighLight should endeavor to cluster "related" data in a segment to improve read performance. The determination of whether data are "related" depends on the particular policy in use. If related data will not fit in the one segment, then their layout on tertiary storage should be arranged to admit a simple prefetch mechanism to reduce further latencies.

Given perfect predictions, policies should migrate data which provides the best benefit to performance (which could mean something like migrating files which will never again be referenced, or referenced after all other files in the cache). Without perfect knowledge, however, migration policies need to estimate the benefits of migrating a file or set of files. We speculate below on some policies that we will evaluate with HighLight.

All the possible policy components discussed below require some additional mechanism support beyond that provided by the basic 4.4BSD LFS. They require some basic migration bookkeeping and data transfer mechanisms, which are described in the next section.

File Size-based Rankings

Earlier studies [3, 12] conclude that file size alone does not work well for selecting files as migration candidates; they recommend using a space-time product (STP) ranking metric (time since last access, raised to a small power, times file size). Strange [16] evaluated different variations on the STP scheme for a typical networked workstation configuration. Those three evaluations considered different environments, but generally agreed on the space-time product as a good metric. Whether these results still work well in the Sequoia environment is something we will evaluate with HighLight.

The space-time product metric has only modest requirements on the mechanisms, needing only the file attributes (available from the base LFS) and a whole-file migration mechanism.

Choosing Block Ranges

In the simplest policies, HighLight could use whole-file migration, with mechanism support based on file access and modification times contained in the inode. However, in some environments whole file migration may be inadequate. In UNIX-like distributed file system environments, most files are accessed sequentially and many of those are read completely [1]. We expect scientific application checkpoints to be read completely and sequentially. In these cases, whole file migration makes sense. However, database files tend to be large, may be accessed randomly and incompletely (depending on the application's queries), and in some systems are never overwritten [13]. Consequently, block-based information is useful, since old, unreferenced data may migrate to tertiary storage while active data remain on secondary storage.

In order to provide migration on a finer grain than whole files, HighLight must keep some information on each disk-resident data block in order to assist the migration decisions. Keeping information for each block on disk would be exorbitantly expensive in terms of space, and often unnecessary. It seems likely that tracking access at a finer grain than whole files can yield a benefit in terms of working set size. Such tracking requires a fair amount of support from the mechanism: access to the sequential block-range information, which implies mechanism-supplied and updated records of file access sequentiality. We do not yet have a clear implementation strategy for this policy.

Namespace Locality

When dealing with a collection of small files, it will be more efficient to migrate several related files at once. We can use a file namespace to identify these collections of "related" files, and migrate directory trees or sub-trees to tertiary storage together. This is useful primarily in an environment where whole subtrees are related and accessed at nearly the same time, such as software development environments. Such a tree could be considered in the aggregate as a file for purposes of applying a migration metric (such as STP).

Assuming such a tree is too large for a single tertiary segment, a natural prefetch policy on a cache miss is to load the missed segment and prefetch remaining segments of the tree cluster.

The primary additional requirement of this policy is a way to examine file system trees without disturbing the access times; this is possible to do with a user program since BSD filesystems do not update directory access times on normal directory accesses, and file inodes may be examined without modification.

Rewriting Cached Segments

It may be the case that data access patterns to tertiary-backed storage will change over time (for example, if several satellite-collected data sets are loaded independently, and then those data sets are analyzed together). Performance may be boosted in such cases by reorganizing the data layout on tertiary storage to reflect the most prevalent access pattern(s) (perhaps to move segments to different tertiary media with access characteristics more suited to those segments). This reorganization can be accomplished by writing cached segments to a new storage location on the tertiary device while the segment is in the cache.

Implicit in this scheme is the need to choose which cached segments should be rewritten to a new location on tertiary storage. All of the questions appropriate to migrating data in the first place are appropriate, so the overhead involved here might be significant (and might be an impediment if cache flushes need to be fast reclaims).

This policy will require additional identifying information on each cache segment to indicate an appropriate locality of reference patterns between segments. Such information could be a segment fetch timestamp or the user-id or process-id responsible for a fetch. Such information could be maintained by the process servicing demand fetch requests and shared with the migrator.

Supporting Migration Policies

To summarize, we can envision uses for (at least) the following mechanism features in an implementation:

- Basic migration bookkeeping (cache lookup control, data movement, etc.)
- Whole-file migration
- Directory and metadata migratable
- Grouping of files by some criterion (namespace)
- Cache fill timestamps/uid/pid
- Sequential block-range data (per-file)

The next section presents the design and implementation of HighLight, which covers many (but not all) of these desired features.

HighLight Design and Implementation

In order to provide "on-line" access to a large data storage capacity, HighLight manages secondary and tertiary storage within the framework of a unified file system based on the 4.4BSD LFS. Our discussion here covers HighLight's basic components, block addressing scheme, secondary and tertiary storage organizations, migration mechanism, and implementation details.

Components

HighLight extends 4.4BSD LFS by adding several new software components:

- A second cleaner, called the *migrator*, which collects data for migration from secondary to tertiary storage
- A disk-resident segment cache to hold read-only copies of tertiary-resident segments and request I/O from the user-level processes, implemented as a pseudo-disk driver
- A pseudo-disk driver which stripes multiple devices into a single logical drive.
- A pair of user-level processes (the service process and the I/O process) to access the tertiary storage devices on behalf of the kernel.

Besides adding these new components, HighLight slightly modifies various portions of the user-level and kernel-level 4.4BSD LFS implementation (such as changing the minimum allocatable block size, adding conditional code based on whether segments are secondary or tertiary storage resident, etc.).

Basic Operation

HighLight implements the normal filesystem operations expected by the 4.4BSD file system switch. When a file is accessed, HighLight fetches the necessary metadata and file data based on the traditional FFS inode's direct and indirect 32-bit block pointers. The block address space appears uniform, so that HighLight just passes the block number to its I/O device driver. The device driver maps the block number to whichever physical device stores the block (a disk, an on-disk cached copy of the block, or a tertiary medium).

The migrator process periodically examines the collection of on-disk file blocks, and decides (based upon some policy) which file data blocks and/or metadata blocks should be migrated to a tertiary medium. Those blocks are then assembled in a "staging segment" addressed by new block numbers assigned to a tertiary medium. The staging segment is assembled on-disk in a dirty cache line, using the same mechanism used by the cleaner to copy live data from an old segment to the current active segment. When the staging segment is filled, the kernel-resident part of the file system requests the server process to copy the dirty line (the entire 1MB segment) to tertiary storage. The request is served asynchronously, so that the migration control policies may choose to move multiple segments in a single logical operation for transfer efficiency.

Disk segments can be used to cache tertiary segments. Since the cached segments are read-only copies of the tertiary-resident version, cache management is relatively simple (involving no write-back issues). As in the normal LFS, when file data are updated, a new copy of the changed data are appended to the current on-disk log segment; the old copy remains undisturbed until its segment is cleaned or ejected from the cache. We don't clean cached segments on disk; any cleaning of tertiary-resident segments would be done directly with the tertiary-resident copy.

If a process requests I/O on a file for which some necessary metadata or file data is not on secondary storage, the cache may satisfy the request. If the segment containing the required data is not in the cache, the kernel requests a demand fetch from the service process and waits for a reply. The service process finds a reusable segment on disk and directs the I/O process to fetch the necessary segment into that segment. When that is complete, the service process registers the new cache line in the cache directory and calls the kernel to restart the file I/O.

The service or I/O process may choose unilaterally to eject or insert new segments into the cache. This allows them to prefetch multiple segments, perhaps based on some policy, hints, or historical access patterns.

Block Addresses

HighLight uses a uniform block address space for all devices in the filesystem. A single HighLight filesystem may span multiple disk and tertiary storage devices. Figure 4 illustrates the mapping of block numbers onto disk (secondary) and tertiary devices. Block addresses can be considered as a pair: (segment number, offset). The segment number determines both the medium (disk device, tape cartridge, or jukebox platter) and the segment's location within the medium. The offset identifies a particular block in that segment.

HighLight allocates a fixed number of segments to each tertiary medium. Since some media may hold a variable amount of data (e.g., due to device-level compression), this number is set to be the

maximum number of segments the medium is expected to hold. HighLight can tolerate device-based compression on tertiary storage since it can keep writing segments to a medium until the drive returns an "end-of-tape" message, at which point the medium is marked full and the last (partially written) segment is re-written onto the next tape. If the compression factor exceeds the expectations, however, all the segments will fit on the tape and some storage at its end may be wasted.

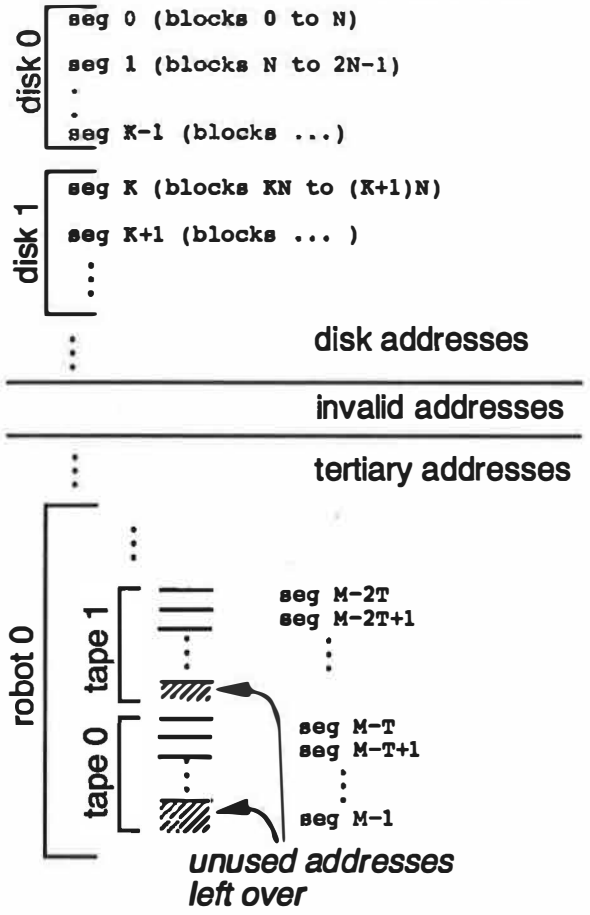


Figure 4: Allocation of block addresses to devices in HighLight

When HighLight's I/O driver receives a block address, it simply compares the address with a table of component sizes and dispatches to the underlying device holding the desired block. Disks are assigned to the bottom of the address space (starting at block number zero), while tertiary storage is assigned to the top (starting at the largest block number). Tertiary media are still addressed with increasing block numbers, however, so that the end of the first medium is at the largest block number, the end of the second medium is just below the beginning of the first medium, etc.

The boundary between tertiary and secondary block addresses may be set at any segment multiple. There will likely be a "dead zone" between valid disk and tertiary addresses; attempts to access these blocks results in an error. In principle, the addition of tertiary or secondary storage is just a matter of claiming part of the dead zone by adjusting the boundaries and expanding the file system's summary tables. However, we do not currently have a tool to make such adjustments after a file system has been created.

We use a single block address space for ease of implementation. By using the same format block numbers as the original LFS, we can use much of its code as is. However, with 32-bit block numbers and 4-kilobyte blocks, we are restricted to less than 16 terabytes of total storage. One segment's worth of address space is unusable for two reasons: (a) we need at least one out-of-band block number ("*-1*") to indicate an unassigned block, and (b) the LFS allocates space for boot blocks at the head of the disk.

We considered using a larger block address and segmenting it into components directly identifying the device, medium, and offset, and using the device field to dispatch to the appropriate device driver. However, the device/medium identity can just as well be extracted implicitly from the block number by an intelligent device driver which is integrated with the cache. The larger block addresses would also have necessitated many more changes to the base LFS, a task which we declined. We considered having the block address include both a secondary and tertiary address, but the difficulty of keeping disk addresses current when blocks are cached (and updating those disk addresses where used) seemed prohibitive. We instead chose to locate the cached copy of a block by querying a simple hash table indexed by segment number.

Using 4-kilobyte blocks necessitates an increased partial segment summary block size (it is only 512 bytes in 4.4BSD LFS). Since the summaries include records describing the partial segment, the larger summary blocks could either reduce or increase overall overhead, depending on whether the summaries are completely filled or not. If the summaries in both the original and new versions are completely full, overhead is reduced with the larger summary blocks. However, the larger summary blocks are almost always too large to be filled in practice, since doing so would require a segment summary to cover an entire segment, and that segment would need to be filled with one block from each of many files. This is possible but not likely given the type of files we expect to find in our environment.

Secondary Storage Organization

The disks are concatenated by a device driver and used as a single LFS file system. Fresh data are written to the tail of the currently-active log

segment. The cleaner reclaims dirty segments by forwarding any live data to the tail of the log. Both the segment selection algorithm, which chooses the next clean segment to be consumed by the log, and the cleaner, which reclaims disk segments, are identical to the 4.4BSD LFS implementations. Unlike the 4.4BSD LFS, though, some of the log segments found on disk are read-only cached segments from tertiary storage.

The ifile, which contains summaries of segments and inode locations, is a superset of that from the 4.4BSD LFS ifile. It has additional flags available for each segment's summary, such as a flag indicating that the segment is being used to cache a tertiary segment and should not be cleaned or overwritten. We also add an indication of how many bytes of storage are available in the segment (which is useful for bookkeeping for a compressing tape or other container with uncertain capacity).

To record summary information for each tertiary medium, HighLight adds a companion file similar to the ifile. It contains tertiary segment summaries in the same format as the secondary segment summaries found in the ifile.

Other special support which a migrator might need to implement its policies can be constructed in additional distinguished files. This might include sequentiality extent data (describing which parts of a file are sequentially accessed) or file clustering data (such as a recording of which files are to migrate together). For efficiency of operation, all the special files used by the base LFS and HighLight are known to the migrator and always remain on disk.

The support necessary for the migration policies may only require user-level support in the migrator, or may involve additional kernel code to record access patterns.

If a need arises for more disk storage, it is possible to initialize a new disk with empty segments and adjust the file system superblock parameters and ifile to incorporate the added disk capacity. If it is necessary to remove a disk from service, its segments can all be cleaned (so that the data is copied to another disk) and marked as having no storage. Tertiary storage may theoretically be added or removed in a similar way.

Tertiary Storage Organization

Tertiary storage in HighLight is viewed as an array of devices each holding an array of media volumes, each of which contains an array of segments. Media are currently consumed one at a time by the migration process. We expect that the migrator may wish to direct several migration streams to different media, but do not support that in our current implementation.

We expect the need for tertiary media cleaning to be rare, because we make efforts to migrate only stable data, and to have available an appropriate spare capacity in our tertiary storage devices. Indeed, the current implementation does not clean tertiary media. We will eventually have a cleaner for tertiary storage, which will clean whole media at a time to minimize the media swap and seek latencies[4]. Since tertiary storage is often very slow (sometimes with access latencies for loading a medium and seeking to the desired offset running over a minute), the relative penalty of taking a bit more access time to the tertiary storage in return for generality and ease of management of the tertiary storage access path is an acceptable tradeoff. Our tertiary storage is accessed via "Footprint", a user-level controller process which uses Sequoia's generic robotic storage interface. It is currently a library linked into the I/O server, but the interface could be implemented by an RPC system to allow the jukebox to be physically located on a machine separate from the file server. This will be important for our environment due to hardware and device driver constraints. Using Footprint also simplifies our utilization of multiple types of tertiary devices, by providing a uniform interface.

Pseudo Devices

HighLight relies heavily on pseudo device drivers, which do not communicate directly with a device but instead provide a device driver *interface* to extended functionality built upon other device drivers and specialized code. For example, a striped disk driver provides a single device interface built on top of several independent disks (by mapping block addresses and calling the drivers for the respective disks).

HighLight uses pseudo device drivers for:

- A striping driver to provide a single block address space for all the disks.
- A block cache driver which sends disk requests down to the striping disk pseudo driver, and which sends tertiary storage requests to either the cache (which then uses the striping driver) or the tertiary storage pseudo driver.
- A tertiary storage driver to pass requests up to the user-level tertiary storage manager.

Figure 5 shows the organization of the layers. The block map driver, segment cache and tertiary driver are fairly tightly coupled for convenience. The block map pseudo-device handles `ioctl()` calls to manipulate the cache and to service kernel I/O requests, and handles `read()` and `write()` calls to provide the I/O server with access to the disk device to copy segments on or off of the disk. To handle a demand fetch request, the tertiary driver simply

[4]Minimizing medium insertion and seek passes is also important, as some tape media become increasingly unreliable after too many readings or too many insertions in tape readers.

enqueues it, wakes up a sleeping service process, and then sleeps as usual for any block I/O. The service process directs the I/O process to fetch the data to disk. When it has been fetched, the service process completes the block I/O by calling into the kernel and restarting the I/O through the cache. It completes like any normal block I/O and wakes up the original process.

User Level Processes

There are three user-level processes used in HighLight that are not present in the regular 4.4BSD LFS: the kernel request service process, the I/O process, and the migrator. The service process waits for requests from either the kernel or from the I/O process: The I/O process may send a status message, while the kernel may request the fetch of a non-resident tertiary segment, the ejection of some cached line (in order to reclaim its space), or the transfer to tertiary storage of a freshly-written tertiary segment.

If the kernel requests a "push" to tertiary storage or a demand fetch, the service process records the request and forwards it to the I/O server. For a demand fetch of a non-resident segment, the service process selects an on-disk segment to act as the cache line. If there are no clean segments available for that use, the service process selects a resident cache line to be ejected and replaced. When the I/O server replies that a fetch is complete, the service process calls the kernel to complete the servicing of the request. The service process interacts with the kernel via `ioctl()` and `select()` calls on a character special device representing the unified block address space.

The I/O server is spawned as a child of the service process. It waits for a request from the service process, executes the request, and replies with a status message. It accesses the tertiary storage device(s) through the Footprint interface, and the on-disk cache directly via the cache raw device. Direct access avoids memory-memory copies and pollution of the block buffer cache with blocks ejected to tertiary storage (of course, after a demand fetch, those needed blocks will eventually end up in the buffer cache). Any necessary raw disk addresses are passed to the I/O server as part of the service process's request.

The I/O server is a separate process primarily to provide for some overlap of I/O with other kernel request servicing. If more overlap is required, the I/O server or service process could be rewritten to farm out the work to several processes or threads to perform overlapping I/O.

The third HighLight-specific process, the migrator, embodies the migration policy of the file system, directing the migration of file blocks to tertiary storage segments. It has direct access to the raw disk device, and may examine disk blocks to inspect inodes, directories, or other structures needed for its policy decisions. It selects file blocks by some

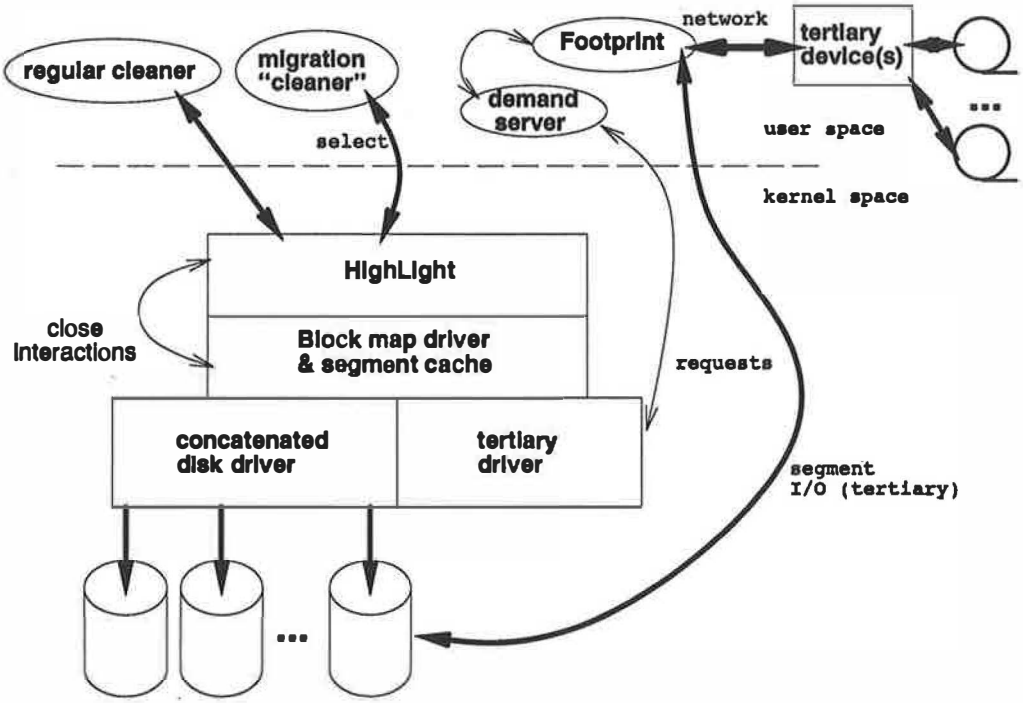


Figure 5: The layered architecture of the HighLight implementation. Heavy lines indicate data or data/control paths; thin lines are control paths only.

criteria, and uses a system call (`lfs_bmapv()`) to find their current location on disk. If they are indeed on disk, it reads them into memory and directs the kernel (via the `lfs_migratev()` call, a variant of the call the regular cleaner uses to move data out of old segments) to gather and rewrite those blocks into the staging segment on disk. Once the staging segment is filled, the kernel posts a request of the service process to copy the segment to tertiary storage.

Performance Micro-benchmarks

To understand and evaluate the performance of HighLight and the impact of our modifications to the basic LFS mechanism, we ran benchmarks with three basic configurations:

1. The basic 4.4BSD LFS.
2. The HighLight version of LFS, using files which have not been migrated.
3. The HighLight version of LFS, using migrated files which are all in the on-disk segment cache.

We ran the tests on an HP 9000/370 CPU with 32 MB of main memory (with 3.2 MB of buffer cache) running 4.4BSD-Alpha. We used a DEC RZ57 SCSI disk drive for our tests, with the on-disk filesystem occupying an 848MB partition. Our tertiary storage device was a SCSI-attached HP 6300 magneto-optic (MO) changer with two drives and 32 cartridges. One drive was allocated for the currently-active writing segment, and the other for reading other platters (the writing drive also fulfilled any read requests for its platter). When running tests with storage To force more frequent medium changes, we constrained HighLight's use of each platter to 40MB (since we didn't have large amounts of data with which to fill the platters to capacity).

Unfortunately, our autochanger device driver does not disconnect from the SCSI bus, and any media swap transactions "hog" the SCSI bus until the robot has finished moving the cartridges. Such media swaps can take many seconds to complete.

Large Object Performance

To test performance with large "objects", we used the benchmark of Stonebraker and Olson [15] to measure I/O performance on relatively large transfers. It starts with a 51.2MB file, considered a collection of 12,500 frames of 4096 bytes each (these could be database data pages, compressed images in an animation, etc). The buffer cache is flushed before each phase of the benchmark. The following operations comprise the benchmark:

- Read 2500 frames sequentially (10MB total)
- Replace 2500 frames sequentially (logically overwrite the old ones)
- Read 250 frames randomly (uniformly distributed over the 12500 total frames, selected

Phase	FFS		Base LFS		HLFS (on-disk)		HLFS (in-cache)	
	time	throughput	time	throughput	time	throughput	time	throughput
10MB sequential read	10.46 s	1002KB/s	12.8 s	819KB/s	12.9 s	813KB/s	12.9 s	813KB/s
10MB sequential write	10.0 s	1024KB/s	16.4 s	639KB/s	17.0 s	617KB/s	17.6 s	596KB/s
1MB random read	6.9 s	152KB/s	6.8 s	154KB/s	6.9 s	152KB/s	7.1 s	148KB/s
1MB random write	3.3 s	315KB/s	1.4 s	749KB/s	1.4 s	749KB/s	1.3 s	807KB/s
1MB read, 80/20 locality	6.9 s	152KB/s	6.8 s	154KB/s	6.9 s	152KB/s	7.1 s	148KB/s
1MB write, 80/20 locality	1.48 s	710KB/s	1.2 s	873KB/s	1.4 s	749KB/s	1.4 s	749KB/s

Table 1: Large Object performance tests. Time values are elapsed times; throughput is calculated from the elapsed time and total data volume. The FFS measurements are from a version with read and write clustering. For the LFS measurements, the disk had sufficient clean segments so that the cleaner did not run during the tests.

File size	FFS access times		HLFS access times			
			in-cache		uncached	
	First byte	Total	First byte	Total	First byte	Total
10KB	0.06 s	0.09 s	0.11 s	0.12 s	3.57 s	3.59 s
100KB	0.06 s	0.27 s	0.11 s	0.27 s	3.59 s	3.73 s
1MB	0.06 s	1.29 s	0.10 s	1.55 s	3.51 s	8.22 s
10MB	0.07 s	11.89 s	0.09 s	13.68 s	3.57 s	44.23 s

Table 2: Access delays for files, in seconds. The time to first byte includes any delays for fetching metadata (such as an inode) from tertiary storage. The FFS measurements are from a version with read and write clustering

with the 4.4BSD `random()` function with the time-of-day as the seed)
- Replace 250 frames randomly
- Read 250 frames with 80/20 locality: 80% of reads are to the sequentially next frame; 20% are to a random next frame.
- Replace 250 frames with 80/20 locality.

Note that for the HighLight version with migrated files, any modifications go to local disk rather than to tertiary storage, so that portions of the file live in cached tertiary segments and other portions in regular disk segments. In practice, our migration policies attempt to avoid this situation by migrating only file blocks which are stable.

Table 1 shows our measurements for the large object test. We were able to test this benchmark on the plain 4.4BSD-Alpha Fast File System (FFS) as well; we used 4096-byte blocks for FFS (the same basic size as used by LFS and HighLight) with the maximum contiguous block count set to 16 (to result in 64-kilobyte transfers in the best case). The base LFS compares unfavorably to the plain FFS; this is most likely due to extra buffer copies performed inside the LFS code. For HighLight, when data have not been migrated to secondary storage, there is a slight performance degradation versus the base LFS (due to the slightly modified system structures). Even when data have been "migrated" but remain cached on disk, the degradation is small.

Access Delays

To measure the delays incurred by a process waiting for file data to be fetched into the cache, we migrated some files, ejected them from the cache, and then read them (so that they were fetched into the cache again). We timed both the access time for the first byte to arrive in user space, and the elapsed time. The files were read from a newly-mounted filesystem (so that no blocks were cached), using the standard I/O library with an 8KB-buffer. The tertiary medium was in the drive when the tests began, so time-to-first-byte does not include the media swap time. Table 2 shows the first-byte and total elapsed times for disk-resident (both HLFS and FFS) and uncached files. FFS is faster to access the first byte, probably because it fetches fewer metadata blocks (LFS needs to consult the inode map to find the file). The time-to-first-byte is fairly even among file sizes, indicating that HighLight does make file blocks available to user space as soon as they are on disk. The total time for the uncached file read of 10MB is somewhat more than the sum of the in-cache time and the required transfer time (computable from the value in Table 5), indicating some inefficiency in the fetch process. The inefficiency probably stems from the extra copies of demand-fetched segments: they are copied from tertiary storage to memory, thence to raw disk, and are finally re-read through the file system and buffer cache. The implementation of this scheme is simple, but performance suffers. A mechanism to transfer blocks directly from the I/O server memory to the buffer cache might provide substantial improvements.

Migrator Throughput

To measure the available bandwidth of the migration path, we took the original 51.2MB file from the large object benchmark and migrated it entirely to tertiary storage, while timing the components of the migration mechanism. The migration path measurements are divided into time spent in the Footprint library routines (which includes any media change or seek as well as transfer to the tertiary storage), time spent in the I/O server main code (copying from the cache disk to memory), and queuing delays. Table 3 shows the measurements; the MO disk transfer rate is the main factor in the performance, resulting in the Footprint library consuming the bulk of the running time.

To get a baseline for comparison with HighLight, we measured the raw device bandwidth available by using dd with the same I/O sizes as HighLight uses (whole segments). We also measured the average time from the start of a medium swap to medium ready for reading. Table 4 shows our raw device measurements.

Phase	Percentage of time consumed
Footprint write	62%
I/O server read	37%
Migrator queueing	1%

Table 3: A breakdown of the components of the archiver/migrator elapsed run times while transferring data from magnetic to magneto-optical (MO) disk

I/O type	Performance
Raw MO read	451KB/s
Raw MO write	204KB/s
Raw RZ57 read	1417KB/s
Raw RZ57 write	989KB/s
Media change	13.5s

Table 4: Raw device measurements. Raw throughput was measured with a set of sequential 1-MB transfers. Media change measures time from an eject command to a completed read of one sector on the MO platter.

Table 5 shows our measurements of two distinct phases of migrator throughput when writing segments to MO disk. The total throughput provided when the magnetic disk is in use simultaneously by the migrator (reading blocks and creating new cached segments) and by the I/O server (copying segments out to tape) is significantly less than the total throughput provided when the only access to the magnetic disk is from the I/O server. When there is no disk arm contention, the I/O server can

write at nearly the full bandwidth of the tertiary medium. The magnetic disk and the optical disk shared the same SCSI bus; both were in use simultaneously for the entire migration process. Since both disks were in use both the disk arm contention and non-contention phases, this suggests that SCSI bandwidth was not the limiting factor and that performance might improve by using a separate disk spindle for the staging cache segments.

Phase	Throughput
Magnetic disk arm contention	111KB/s
No arm contention	192KB/s
Overall	135KB/s

Table 5: Migrator throughput measurements for phases with and without disk arm contention.

Conclusions

Sequoia 2000 needs support for easy access to large volumes of data which won't economically fit on current disks or file systems. We have constructed HighLight as an extended 4.4BSD LFS. It manages tertiary storage and integrates it into the filesystem, with a disk cache to speed its operation. The mechanisms provided by HighLight are sufficient to support a variety of potential migration control policies, and provide a good testbed for evaluating these policies. The performance of HighLight's basic mechanism when all blocks reside on disk is nearly as good as the basic 4.4BSD LFS performance. Transfers to magneto-optical tertiary storage can run at nearly the tertiary device transfer speed.

We intend to evaluate our candidate policies to determine which one(s) seem to provide the best performance in the Sequoia environment. However, it seems clear that the file access characteristics of a site will be the prime determinant of a good policy. Sequoia's environment may differ sufficiently from others' environments that direct application of previous results may not be appropriate. Our architecture is flexible enough to admit implementation of a good policy for any particular site.

Future Work

To avoid eventual exhaustion of tertiary storage, HighLight will need a tertiary cleaning mechanism that examines tertiary volumes, a task which would best be done with at least two access points to avoid having to swap between the being-cleaned medium and the destination medium.

Some other tertiary storage systems do not cache tertiary resident files on first reference, but bypass the cache and return the file data directly. A second reference soon thereafter results in the file being cached. While this is less feasible to implement directly in a segment-based migration scheme, we could designate some subset of the on-disk cache lines as "least-worthy" and eject them first upon reading a new segment. Upon repeated access the cache line would be marked as part of the regular pool for replacement policy (this is essentially a cross between a nearly-MRU cache replacement policy and whatever other policy is in use).

As mentioned above, the ability to add (and perhaps remove) disks and tertiary media while online may be quite useful to allow incremental growth or resource reallocation. Constructing such a facility should be fairly straightforward.

There are a couple of reliability issues worthy of study: backup and media failure robustness. Backing up a large storage system such as HighLight would be a daunting effort. Some variety of replication would likely be easier (perhaps having the Footprint server keep two copies of everything written to it). For reliability purposes in the face of a medium failure, it may be wise to keep certain metadata on disk and back them up regularly, rather than migrate them to a potentially faulty tertiary medium. Doing so might avoid the need to examine all the tertiary media in order to reconstruct the filesystem after a tertiary medium failure.

Code Availability

When the system is robust enough for external distribution, source code for HighLight will be available to licensees of 4.4BSD. Contact Project Sequoia 2000 at the University of California, Berkeley if you wish to obtain the code.

Acknowledgments

The authors are grateful to students and faculty in the CS division at Berkeley for their comments on early drafts of this paper. An informal study group on Sequoia data storage needs provided many keen insights we incorporated into our work. Ann Drapeau helped us understand some reliability and performance characteristics of tape devices. Randy Wang implemented the basic Footprint interface.

We are especially grateful to Mendel Rosenblum and John Ousterhout whose LFS work underlies this project, and to Margo Seltzer and the Computer Systems Research Group at Berkeley for implementing 4.4BSD LFS.

Bibliography

[1] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a Distributed File System. *Operating Systems Review*, 25(5):198-212, October 1991.

[2] General Atomics/DISCOS Division. The Uni-Tree Virtual Disk System: An Overview.

Technical report available from DISCOS, P.O. Box 85608, San Diego, CA 92186, 1991.

[3] D. H. Lawrie, J. M. Randal, and R. R. Barton. Experiments with Automatic File Migration. *IEEE Computer*, 15(7):45-55, July 1982.

[4] Marshall Kirk McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181-197, August 1984.

[5] Ethan L. Miller, Randy H. Katz, and Stephen Strange. An Analysis of File Migration in a UNIX Supercomputing Environment. In *USENIX Association Winter 1993 Conference Proceedings*, San Diego, CA, January 1993. The USENIX Association.

[6] James W. Mott-Smith. The Jaquith Archive Server. UCB/CSD Report 92-701, University of California, Berkeley, Berkeley, CA, September 1992.

[7] Michael Olson. The Design and Implementation of the Inversion File System. In *USENIX Association Winter 1993 Conference Proceedings*, San Diego, CA, January 1993. The USENIX Association.

[8] Sean Quinlan. A Cached WORM File System. *Software–Practice and Experience*, 21(12):1289-1299, December 1991.

[9] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *Operating Systems Review*, 25(5):1-15, October 1991.

[10] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. An Implementation of a Log-structured File System for UNIX. In *USENIX Association Winter 1993 Conference Proceedings*, San Diego, CA, January 1993. The USENIX Association.

[11] Alan Jay Smith. Analysis of Long-Term File Reference Patterns for Application to File Migration Algorithms. *IEEE Transactions on Software Engineering*, SE-7(4):403-417, 1981.

[12] Alan Jay Smith. Long-Term File Migration: Development and Evaluation of Algorithms. *Communications of the ACM*, 24(8):521-532, August 1981.

[13] Michael Stonebraker. The POSTGRES Storage System. In *Proceedings of the 1987 VLDBConference*, Brighton, England, September 1987.

[14] Michael Stonebraker. An Overview of the Sequoia 2000 Project. Technical Report 91/5, University of California, Project Sequoia, December 1991.

[15] Michael Stonebraker and Michael Olson. Large Object Support in POSTGRES. In *Proc. 9th Int'l Conf. on Data Engineering*, Vienna, Austria, April 1993. To appear.

[16] Stephen Strange. Analysis of Long-Term UNIX File Access Patterns for Application to Automatic File Migration Strategies. UCB/CSD Report 92-700, University of California, Berkeley, Berkeley, CA, August 1992.

Author Information

John Kohl is a Software Engineer with Digital Equipment Corporation. He holds an MS in Computer Science from the University of California, Berkeley (December 1992) and a BS in Computer Science and Engineering from the Massachusetts Institute of Technology (May 1988). He worked several years at MIT's Project Athena, where he led the Kerberos V5 development effort. He may be reached by e-mail at jtkohl@cs.berkeley.edu, or by surface mail at Digital Equipment Corporation, ZKO3-3/U14, 110 Spit Brook Road, Nashua, NH 03062-2698.

Carl Staelin works for Hewlett-Packard Laboratories in the Berkeley Science Center and the Concurrent Systems Project. His research interests include high performance file system design, and tertiary storage file systems. As part of the Science Center he is currently working with Project Sequoia at the University of California at Berkeley. He received his PhD in Computer Science from Princeton University in 1992 in high performance file system design. He may be reached by e-mail at staelin@hpl.hp.com, or by telephone at (415) 857-6823, or by surface mail at Hewlett-Packard Laboratories, Mail Stop 1U-13, 1501 Page Mill Road, Palo Alto, CA 94303.

Michael Stonebraker is a professor of Electrical Engineering and Computer Science at the University of California, Berkeley, where he has been employed since 1971. He was one of the principal architects of the INGRES relational data base management system which was developed during the period 1973-77. Subsequently, he constructed Distributed INGRES, one of the first working distributed data base systems. Now (in conjunction with Professor Lawrence A. Rowe) he is developing a next generation DBMS, POSTGRES, that can effectively manage not only data but also objects and rules as well.

He is a founder of INGRES Corp (now the INGRES Products Division of ASK Computer Systems), a past chairman of the ACM Special Interest Group on Management of Data, and the author of many papers on DBMS technology. He lectures widely, has been the keynote speaker at recent IEEE Data Engineering, OOPSLA, and Expert DBMS Conferences, and has been a consultant for several organizations including Citicorp, McDonnell-Douglas, and Pacific Telesis. He may be reached by e-mail at mike@cs.berkeley.edu or surface mail at 549 Evans Hall, Berkeley, CA 94720.

An OSF/1 UNIX for Massively Parallel Multicomputers

Roman Zajcew, Paul Roy, David Black, Chris Peak, Paulo Guedes, Bradford Kemp, John LoVerso, Michael Leibensperger, Michael Barnett, Faramarz Rabii, & Durriya Netterwala
– OSF Research Institute and Locus Computing Corporation

ABSTRACT

This paper describes the architecture and implementation of a version of the OSF/1 Unix operating system designed to run on multicomputer hardware platforms. The multicomputer hardware platforms targeted can consist of hundreds or even thousands of individual nodes, where each node consists of one or more processors.

The multicomputer version of OSF/1 Unix (called OSF/1 AD TNC) is built on the Mach 3.0 Microkernel and the OSF/1 MK Single Server. These have been modified to run in the multicomputer environment and provide a view of the hardware that looks like a conventional but massively scaled up shared memory multiprocessor. The operating system presents this notion of a Single System Image by building Unix functionality on top of base Mach services running on each node in the multicomputer.

The focus of this paper is on the particular enhancements made to standard OSF/1 functionality to operate in a multicomputer environment without incurring system bottlenecks. These include a new distributed file system, a distributed implementation of sockets, and enhancements to process management functionality to support remote processing and load leveling. Extensions to the operating system interface to allow users to take advantage of the parallelism of the multicomputer hardware are also discussed.

Introduction

Historically, hardware vendors have two strategies that can be used individually or in combination to increase performance: (a) increase processor performance and (b) utilize parallelism.

The parallelism architecture that has received the most attention in the recent past has been SMP (Symmetric MultiProcessing), which is a shared-memory, UMA (Uniform Memory Access) architecture. The SMP architecture is attractive because the operating system can be multithreaded to run on multiple processors. Because of the global shared memory, user processes can simply run on any processor that is available. Unfortunately, SMP systems do not scale past tens of processors.

Another type of parallel processing is MPP (Massively Parallel Processing). MPP computers, also known as *multicomputers*, can consist of hundreds or even thousands of *nodes* (each node consisting of one or more processors) connected via a high-speed interconnect. A multicomputer in which each node is able to access local memory only is known as a NORMA (NO Remote Memory Access) computer. A multicomputer with the additional ability to access memory in other nodes is known as a NUMA (Non Uniform Memory Access) computer. Multicomputers of both types are currently being built and marketed by several manufacturers.

In this paper, we are primarily addressing the concerns of NORMA multicomputer systems. Many of the software concepts described in this paper would apply to NUMA multicomputer systems, but the reference implementation is for a NORMA multicomputer system. For the rest of this paper, when "MPP systems" or "multicomputers" are referenced, it is "NORMA multicomputer systems" that are actually being referred to.

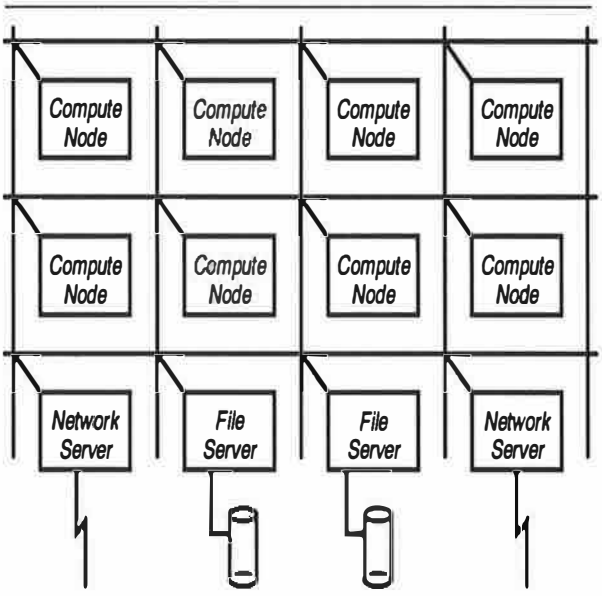


Figure 1: Multicomputer Architecture

In a typical multicomputer system (see Figure 1), the nodes of the multicomputer are divided into three groups: nodes used for input/output and connectivity (I/O nodes or file server nodes); nodes dedicated to parallel applications (compute nodes); and nodes for interactive use (service nodes). The number of nodes in each division varies from system to system – indeed, the division between service nodes and compute nodes may vary depending upon the time of day.

Although the hardware of multicomputers is inherently scalable, a problem has been the provision of a scalable, distributed operating system. While proprietary operating systems for multicomputers have been delivered in the past, current wisdom dictates the necessity for an open operating system and Application Programming Interface (API). Hence, a version of OSF/1 that runs on multicomputers has been developed.

This paper describes OSF/1 AD TNC[1], the multicomputer version of OSF/1 that makes a multicomputer appear as a massively scaled up SMP computer, while avoiding OS bottlenecks. OSF/1 AD TNC was produced by the Open Software Foundation and Locus Computing Corporation, and is based on an enhanced version of the OSF/1 MK Single Server, which is itself based on the OSF/1 operating system and the Mach 3.0 microkernel.

After providing background information on technology incorporated into OSF/1 AD TNC, the following sections describe its goals, architecture, and detailed design. The paper concludes with sections on related work, implementation status, and planned future enhancements.

Background

Mach 3.0 Microkernel

Mach provides a flexible execution environment for both system and user applications. It exposes the management of CPU, communication, virtual memory, and secondary storage resources in a manner that allows system applications to make effective use of these resources. The important features of Mach are:

- *Task and Thread Management*: Mach supports the task and thread abstractions for execution management. A task is a passive resource abstraction, consisting of an address space and communication access to system and server facilities. Computation within a task is performed by one or more threads that fully share the address space and all other resources of the task. Thread scheduling to processors is controlled by the Mach kernel, and threads can execute in parallel on shared memory multiprocessors. Both timesharing and fixed priority policies are supported.
- *Interprocess Communication*: Mach provides interprocess communication via ports and messages. This is actually inter-task communication, but is known as IPC for historical reasons. Ports are protected communication endpoints; only Mach tasks with appropriate capabilities (known as port rights) may send or receive messages on a port (at most one task may receive, but multiple tasks may send). All services, resources, and facilities exported by the Mach kernel and servers are represented by ports. For example, Mach tasks and threads are manipulated by sending messages to ports that represent them.
- *Memory Object Management*: The address space of a Mach task is represented as a collection of mappings from addresses to offsets within memory objects. A Mach memory object represents a single source of memory (e.g., the file from which an executing program was loaded). The kernel manages physical memory as a cache of memory object contents; access to the actual memory (i.e., backing storage) is via a Mach port to which messages can be sent containing data or requesting that it be supplied [Young 87]. This allows memory objects to be implemented by user-state programs such as a file system server or database application.
- *System Call Redirection*: The Mach kernel allows system call traps to be handled in user mode by code executing in the same task. This supports binary compatibility with existing applications without inserting additional code (e.g., from Unix) into the kernel. Similar facilities are provided for redirecting exceptions to user mode handlers [Black 88].
- *Device Support*: The Mach kernel provides low-level device support [Forin 91]. Each device is represented by a port to which messages can be sent to transfer data and control the device. The request and reply messages for the read and write operations are exported as distinct interfaces, supporting both synchronous and asynchronous I/O interactions.

Mach and Multicomputers

Mach supports multicomputer systems by transparently extending communication and memory management services across the multicomputer's interconnection network. Mach's communication mechanisms are location transparent, and can be extended across local area networks by user mode communication servers [Sansom 86]. By comparison to local area networks, well-designed multicomputer interconnects have order of magnitude or more

[1] OSF/1 AD TNC is an acronym for OSF/1 with Advanced Development (AD) extensions from OSF Research Institute and Transparent Network Computing (TNC) extensions from Locus Computing Corp.

increases in bandwidth and reduction of hardware and software access latency [Intel 92]. The multicomputer extension of Mach IPC [Barrera 91] (known as NORMA IPC) is implemented in the Mach kernel to take advantage of the interconnect performance, and allow direct communication among Mach kernels on different multicomputer nodes (each node runs a separate Mach kernel). This IPC extension is completely transparent to Mach tasks; the locations of the communicating tasks (same or different nodes) do not affect the behavior.

The memory management extensions for multicomputers are motivated by practical concerns. Mach allows memory object managers to support object access by more than one kernel and manage the resulting distributed consistency. As a result, NORMA IPC is sufficient to support multicomputer systems implementation. However, in practice, most memory object managers do not implement multikernel memory object consistency, usually for complexity reasons. Such managers do not work correctly on a multicomputer, and situations in which they malfunction are more common than one might expect. Although actual use of shared memory is quite rare, lazy evaluation of inherited memory (e.g., across a `fork()`) causes multiple nodes to access the same memory when the child task is created on a node other than the parent's.

To support the use of arbitrary Mach memory object managers on a multicomputer system, the XMM (eXtended Memory Management) subsystem has been added to the Mach kernel [Barrera 93]. Like NORMA IPC, XMM is part of every Mach kernel on a multicomputer; its implementation of distributed shared memory makes the collection of Mach kernels on a multicomputer behave as if they were a single kernel when communicating with memory object managers. This removes the complexity of distributed shared memory functionality from the memory object managers. XMM also includes support for copy on reference between nodes for lazy evaluation of memory inherited between tasks on different nodes; this support is used only for memory managed by memory management interfaces, as NORMA IPC does not participate in copy on reference.

OSF/1 Operating System

The Open Software Foundation's OSF/1 is a standards-compliant open operating system that incorporates advanced features while providing compatibility with industry standards and support for existing applications. The advanced features include support for shared memory multiprocessors, B1-level security, logical volume management, multithreaded applications, dynamic system configuration via loadable kernel modules, internationalization and localization (foreign language and culture support). Compatibility is provided to interface specifications from the System V and POSIX collections, as well as for the 4.3BSD programming environment. Further details can be found in [OSF 90] [OReilly 91].

OSF/1 is based on the Mach 2.5 operating system from Carnegie Mellon University. Both are integrated or monolithic kernel systems that implement the majority of system functionality in the operating system kernel (but not all, for example OSF/1's program loader runs in user space). Their kernels contain the core Mach technology and additional Unix functionality. OSF/1 has significantly upgraded the 4.3BSD portion of Mach 2.5 to implement advanced features and obtain compliance with standards.

OSF/1 MK Single Server

OSF/1 AD TNC is based on the OSF/1 MK Single Server. The OSF/1 MK Single server was created by dividing the OSF/1 monolithic kernel into the Mach 3.0 kernel and the OSF/1 MK Single Server. This was achieved by replacing OSF/1's Mach 2.5 internals with Mach 3.0, and adding a layer of compatibility code to allow the rest of OSF/1 to execute as a user-mode server (the OSF/1 MK server). Among the features of the compatibility code are a threads library that provides lightweight user threads on top of Mach's kernel threads, and an emulation library (called the *emulator*) that implements some system functionality in application address spaces. This work was based on an earlier conversion effort at CMU involving the Mach 2.5 system [Golub 90]. The resulting OSF/1 MK server is pagable, preemptible, and multithreaded (leveraging OSF/1's support for symmetric multiprocessors).

Using OSF/1 MK Single Server as the base for OSF/1 AD TNC simplifies the transition to a multicomputer by expressing all of OSF/1's features in terms of Mach's location transparent kernel abstractions.

The OSF/1 MK system is based on a high level mapping of the Unix process model to Mach abstractions. The server is represented to a process by a single port (used for all service requests), and this port serves to identify the requesting process to the server. A system call trap executed by the application is redirected to an emulation library in the application's address space. This library converts most system calls to service request messages and sends them to the server. A shared memory window between the server and each application is used to optimize some data transfers in a fashion similar to (but more primitive than) LRPC [Bershad 90].

Goals

The primary goal of OSF/1 AD TNC was to build a distributed operating system that enables users to take advantage of multicomputer hardware, while retaining full OSF/1 (and thus Unix) semantics. This means that:

- There is a single file name space that spans

all the nodes, and access to all files retains *single system image* semantics.
- Scalability bottlenecks are avoided in the operating system, by distributing control of the following OS subsystems:
  - file system
  - socket protocol stacks
  - process management.
- Processes executing on any node of the multicomputer have access to all resources and facilities of OSF/1, as if they were executing on an SMP architecture (including maintaining shared memory between nodes).
- Existing applications can take advantage of all the computational resources of the multicomputer via automatic load leveling, while at the same time specially parallelized programs can be explicit in their use of multiple nodes.
- Full compatibility with OSF/1 1.0 commands and libraries is maintained (implying compatibility with XPG/3, POSIX 1003.1, SVID Issue 2, and with other standards).

The initial performance goal is to approximate the performance of the OSF/1 MK system, after taking into account inherent discrepancies due to the multicomputer architecture (such as the fact that peripherals are remote). This goal will be refined and expanded as experience and measurements are acquired.

Architecture

The operating system architecture is derived directly from the target hardware architecture, in which large numbers of nodes are connected via a high-speed interconnect, and some subset of the nodes have peripheral devices attached. The OS must allow all nodes and devices to be efficiently utilized, and in particular, must distribute OS functionality in order to avoid bottlenecks.

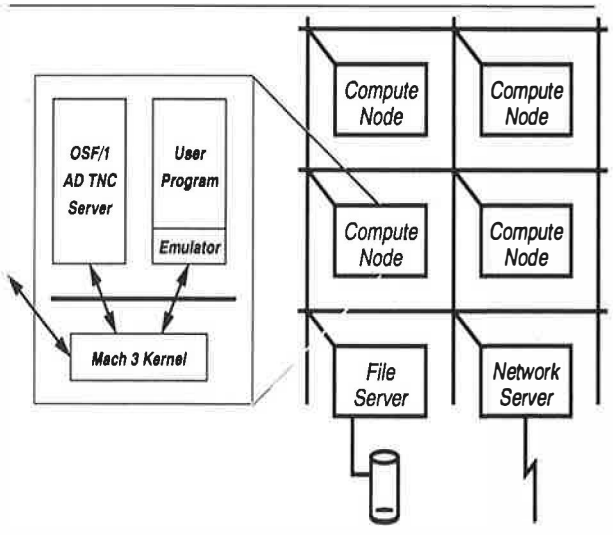


Figure 2: OSF/1 AD TNC Architecture

This leads to the following architectural model (see Figure 2):
- The Mach microkernel runs on all nodes of the multicomputer, providing generic task/thread management, memory management, communication services, and device access.
- Specialized user-space servers implement Unix functionality, such as file service, process management, and networking.
- Disk and networking devices are managed by servers typically co-located on the same node with the device (although co-location is not mandatory).
- Process management functionality is distributed across most or all nodes on which application processes are run.
- An *emulator* (the previously mentioned emulation library) [Golub 90] [Julin 91] is available in each process's address space to provide some Unix functionality and perform system call-to-message conversion. It also contains a thread to receive callback messages from servers in support of interruptible system calls and file caching.
- Process management system calls that require client-server interaction are converted by the emulator to messages to the process management server, which is normally co-located on the same node as the process. File management system calls that require client-server interaction are converted to messages to the file server that is managing the particular file being manipulated.

This model served as the basis for implementing OSF/1 AD TNC, as described in the following sections.

Detailed Design

In order to meet the above goals, the base OSF/1 Unix code was enhanced in the following ways:
- The file system was modified to support the integration of multiple file servers into a coherent whole, providing a single file system name space, location transparency, and remote device handling. Typical file system-related system calls require at most one client-server interaction. Extensive caching is used to allow many frequently used operations to require no client-server interaction.
- Control of socket protocol stacks is distributed across multiple nodes. This was done in a way to minimally impact the existing code in the protocol stacks.
- Process management is distributed across all nodes of the multicomputer. Many system calls are completely serviced by the server on the node local to the issuing process without

necessitating internode communication. However, system calls affecting process relationships spanning multiple nodes (for example, signaling a process on another node) is conducted in a way transparent to all user processes.

Simple primitives were added to allow programs to explicitly use multiple nodes. Automatic load leveling to utilize all the nodes of the multicomputer was also added.

File System Enhancements

Within the multicomputer, certain nodes are used as file servers. Each file server provides file service for one or more file systems (partitions). Mount operations are used to assemble multiple file systems across multiple file servers into a single file name space. Access to all files within the name space, including to remote devices, is location transparent. Since devices can reside anywhere in the name space, device naming is separate from device location.

To achieve high levels of performance, extensive distributed caching of file data is done within the context of the user process, so that some frequently used system calls do not require client-server interaction. This caching makes extensive use of the Mach memory object abstraction. For file-related system calls that cannot be handled by the distributed cache, typically at most one client-server interaction is required.

As the file server for a file and the process management server for a process may be on separate nodes, a *credentials service* was implemented to allow state to be exchanged between file servers and process managers. The fact that file-related system calls may be processed on a node other than the process management node also necessitated a special design for file-related system calls interrupted by signals.

File Servers

File servers are user-space servers that provide access to UFS file systems (the NFS Support section discusses NFS access). Each node may have *at most one* file server, and all of a node's (file system) devices are managed by *exactly one* file server. Typically, the file server for a node's devices will run on that node, but it may also run on a different node (because of Mach's network transparent device interface).

Distributed File Name Space

Construction of the file name space is based on the mount model, whereby a file system (partition) is mounted on top of a directory. A file system and the directory that it's mounted on need not be co-located on the same node, thus allowing file systems managed by multiple file server nodes to be assembled into a single file name space. The resulting behavior is exactly as expected from a standard OSF/1 system where a (local) file system is mounted on a (local) directory. In other words, *single system image* semantics are maintained. An example file name space is shown in Figure 3.

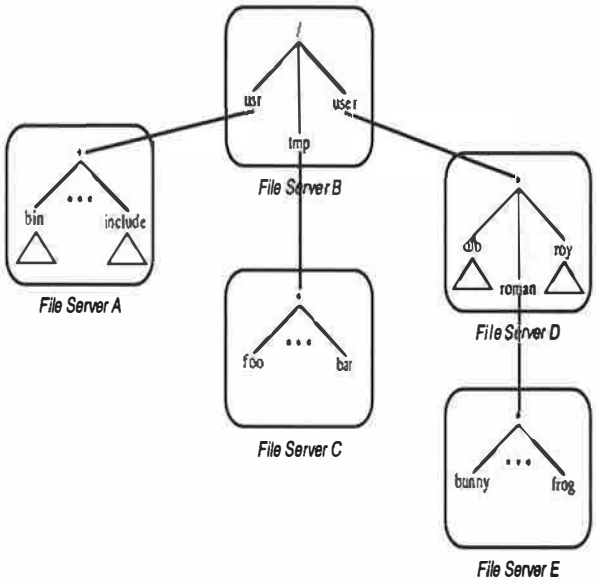


Figure 3: File Name Space

As in standard OSF/1, file systems are identified by device special files in the name space. However, in a multicomputer environment another piece of information is required: the *node* to which the device partition is attached. This is specified by a node number field in the special file's inode. It is set using a new command `rmknod` that is identical to `mknod`, but accepts the node number as an additional argument. This node number is used by the `mount` operation to construct remote mounts.

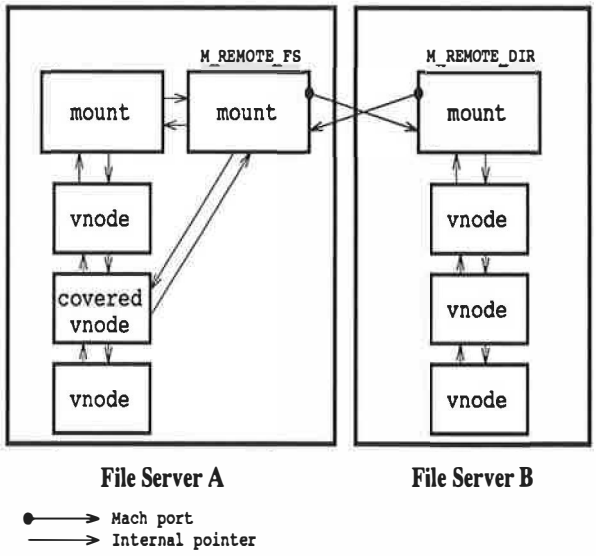


Figure 4: Remote Mount Data Structures

A remote mount is represented by a pair of mount structures, as shown in Figure 4. A local structure resides in the file server managing the covered directory and a remote structure resides in the file server for the mounted file system. The local mount structure contains a flag indicating it represents a remote mount, and a send right to the port associated with the remote mount structure. The real information for the mount is contained in the remote mount structure. It also contains a send right to the port associated with the local mount structure in order to support pathname traversal of "..".

The mount() operation is targeted at the file server managing the directory being mounted on. Based on the node information stored in the special file representing the file system being mounted, the mount code determines whether a remote mount needs to be set up. If so, a mount structure is created and marked as M_REMOTE_FS (meaning the file system is remote), a port representing it is allocated, and a request containing a send right to this port is sent to the remote file server. Upon receipt, the remote file server creates a mount structure, marking it M_REMOTE_DIR (meaning the directory is remote), and stores the send right received as a parameter in the m_remoteport field. A port representing the new structure is allocated and a send right to it is returned to the original server, which then completes the operation by storing it in its m_remoteport field. (Further information on remote mounts may be found in [Paciorek 92a].)

Pathname translation works exactly as in standard OSF/1 (including pathname caching), until a remote mount point is encountered. At that point, the remainder of the pathname is forwarded to the remote server so that translation may continue. This process is repeated until the pathname is fully translated.

Client Access to the File System

As in standard OSF/1, processes access files via system calls. In a multicomputer environment, it is extremely important to implement these system calls in a manner that minimizes IPC messages and server context switches. OSF/1 AD TNC accomplishes this using the Mach system call redirection mechanism (described in the Background Section) to implement functionality in a per-process *emulator* residing in each process's address space. In particular, the emulator:

- contains the per-process file descriptor table, where each table entry has a port right mapping to the open file structure on a file server. Hence, operations on open files communicate with at most one server.
- contains send rights to the ports representing the root and current working directories for the process. Hence, most (but not all) operations on pathnames (e.g., open(), mkdir()) communicate with one server.
- provides a location to implement mapped access to open files and accrue the benefits of node-local file data caching (see the next section).

File servers export open file and directory vnode data structures by associating ports with them and providing clients with send rights. Messages received on ports are then converted to function calls on the corresponding data structures.

System calls that perform pathname translation are executed using a *remote system call* model; i.e., messages to file servers instruct the server to resolve the pathname and execute the operation. If, during pathname translation, a remote mount is encountered, the system call is forwarded to the file server servicing the mount, which then continues the translation. This process is repeated until the pathname is fully translated (to a corresponding vnode), at which time the operation is executed.

The emulator determines whether to send an initial system call message to the root or current working directory port depending on whether the pathname is absolute or relative. This effectively acts as a two-entry cache, but can still result in server-to-server messages when remote mount points are crossed during pathname translation. A planned enhancement is to expand this caching mechanism to include a pathname prefix table [Cheriton 89] [Julin 91] [Welch 86].

An alternative to the remote system call model would have been to resolve the pathname (to a vnode) and execute the system call as separate operations (analogous to the VFS interface internal to Unix [Kleiman 86]). However, avoiding extra messages due to pathname translation would have required a pathname caching scheme that was deemed infeasible within the context of the emulator.

The open() system call executes as described above, but in addition a send right for the open file structure is returned to the emulator. The emulator stores this right in its file descriptor table so that subsequent system calls for the open file are sent directly to the proper file server.

The file descriptor table is implemented such that a dup() system call effectively increments an internal data structure reference without contacting a server. When a process fork()'s, the emulator is automatically duplicated (copy-on-write) in the child address space (including the file descriptor table data structures), but send rights to open files must be explicitly inserted into the child process using the Mach kernel interface mach_port_insert_rights() [Loepere 92]. Access to a shared file offset between parent and child processes is not a problem, because the offset is stored in the common file structure within the file

server (except for Unix regular files which are managed as described in the next section).

Closing files is accomplished using Mach's *no-more-senders* notification facility [Draves 90]. File servers request that they be notified (by the kernel) when no more send rights exist to an open file structure. At `close()` time, the emulator simply deallocates the send right it has associated with the open file. When all such rights have been deallocated (meaning no clients have access to the open file structure) the file server will receive a notification and perform tear down as necessary.

A more detailed description of the information presented in this section may be found in [Paciorek 92a] [Paciorek 92b].

Unix File Caching via Mach Memory Objects

The Mach kernel manages physical memory as a cache of memory object contents; access to the actual memory (i.e., backing storage) is via a Mach port (known as a *memory object port*) to which messages can be sent containing data or requesting that it be supplied. This allows memory objects to be implemented by user-state programs such as a file system server or database application.

OSF/1 AD TNC uses Mach memory objects to cache data from Unix regular files. File servers provide backing storage for memory objects using the standard OSF/1 Unix File System (UFS). Unix `read()` and `write()` system calls are converted to mapped accesses to the corresponding memory objects. Cache hits to already-resident data are satisfied without file server interaction. File servers provide *external memory management* by interacting with the Mach kernel via a protocol known as the *External Memory Management Interface* [Young 87] [Young 89]. This protocol is used to fulfill cache misses, clean pages on behalf of `sync()` , flush pages on behalf of `truncate()`, etc.

The Mach 3.0 4.3BSD Single Server and Chorus SVR4 systems [Dean 92] [Batlivala 92] have implemented similar file caching schemes for single node systems. OSF/1 AD TNC operates in a multicomputer environment using Mach's NORMA XMM kernel subsystem (described earlier) to implement cache management across nodes (i.e., distributed shared memory). Accesses to mapped memory objects are performed by the per-process emulator. A file's memory object port is obtained at `open()` time and stored in the emulator's file descriptor table.

Implementing POSIX file I/O semantics [IEEE 90] requires `read()/write()` atomicity and proper update of a file's accessed and modified times. OSF/1 AD TNC implements these via a message-based token protocol between emulators and file servers. A token is associated with each open file and guarantees multi-reader/single-writer access to the file's:

- data, including its length
- seek pointer
- accessed and modified status.

Access to a file's data and the file's accessed and modified status are synchronized across all processes accessing the file, whereas access to a seek pointer is synchronized only across the processes sharing an open file.

A token (and a file's length and seek pointer) is obtained in a file server's reply to an open request, and is stored in the file descriptor table. When a process executes a `read()` or `write()` system call, its emulator first checks that it still has the token. If so, it proceeds to access the memory object at the offset specified by the seek pointer (or, at the end-of-file if it's an append-mode `write()`). The emulator code must also prohibit reading beyond the end-of-file.

After a successful access, the emulator updates the file's seek pointer, the file's length (if necessary), whether the file was read (accessed) or written (modified), and stores the information in the file descriptor table.

Subsequent I/O's may find the token still held and proceed as above. However, it is also possible the token has been revoked, in which case a message to the file server is required to reacquire a token.

A file server revokes a token by sending a message to an emulator's callback thread, instructing it to release the token by performing a "write-back" of the cached state. A token is revoked either because another process is attempting to acquire the token on behalf of a `read()`, `write()`, or `lseek()`, or because the file server needs the token on behalf of some other operation. For example, `truncate()` requires mutually exclusive access to the file, `sync()` must determine if there are dirty pages to be cleaned, and `stat()` must determine whether the file has been accessed or modified so that the proper times may be reported.

Note that POSIX semantics for updating a file's accessed and modified times only require that they be updated during the last `close()` of the file, or if the file's attributes are read (due to a `stat()` or `fstat()`). Thus, emulators retain booleans indicating whether a file has been accessed or modified, relying on file servers to revoke tokens and update the times as necessary.

If a file with a token is `close()`'d, the emulator sends a message to the file server releasing the token and its associated state.

Tokens are implemented using Mach ports. If a process aborts abnormally, file servers will receive *no-more-senders* notifications for all outstanding tokens, allowing them to perform necessary garbage collection.

The bulk of the code implementing the token mechanism resides in a file server module known as the *mapped files module*. It exports message-based interfaces for emulators as well as internal interfaces for cache management and synchronization. The standard OSF/1 file system code has been modified to invoke these interfaces, with most of the changes isolated to the Unix File System (UFS). Fortunately, these changes are layered in such a way that they can eventually be implemented in a separate file system stacked on top of the UFS.

Remote Device Handling

In standard OSF/1, devices such as tty's and disk partitions are identified via device special files in the file name space. As described previously, a multicomputer environment must have knowledge of the node to which a given device is attached, and hence node numbers are stored in device special files' inodes.

The problem becomes one of efficiency. In particular, if the file server managing a device's special file is also managing the actual device then it becomes a bottleneck because most special files reside in the `/dev` directory. In addition, it is desirable for a file server to be co-located on the same node with the devices it's managing.

OSF/1 AD TNC solves this problem by allowing different file servers to manage a device and the special file representing it. The result is that accesses to an open device communicate only with the file server managing it.

As for `open()`'ing a device, the pathname is resolved to the file server managing the special file, which then forwards an open request to the file server managing the device, passing along a port send right to be used for subsequent communication back to the file server managing the special file. The device is then opened and a port send right to the file structure is returned to the process's emulator, which stores it in its file descriptor table. Thus, at completion of `open()` processing, the emulator has a direct connection to the file server managing the device, which in turn has a connection to the file server managing the special file.

In order to maintain correct accessed and modified times in a special file's inode, the system relies on POSIX semantics [IEEE 90], which only requires that the times be updated during the last `close()` of the device, or if the device's attributes are read (due to a `stat()` or `fstat()`). Thus, a file server managing an open device retains booleans indicating whether a device has been accessed or modified. This information is propagated back to the file server managing the special file (using the connection established at `open()` time) either via *push* (at last `close()` time) or *pull* (at `stat()` time).

`fstat()` requires contact with both file servers (a request message is sent to the file server managing the device, which must contact the file server managing the special file to obtain all the attributes). Avoiding this message would require full caching of a special file's inode in the file server managing the device (as is done in [Batlivala 92]), but the payoff was not deemed worth the extra complexity.

Credentials Service

In a monolithic Unix system, process management and file system functionality co-reside in the same address space. In OSF/1 AD TNC, the functionality may be partitioned across multiple address spaces, perhaps residing on different nodes in the multicomputer. The credentials service provides a mechanism for bidirectional exchange of information between process managers and file servers. In particular, the credentials service supports the propagation of per-process information from:

- process managers to file servers, including process id, process group id, session id, credentials, file creation mask, and file size limits.
- file servers back to process managers, currently only the file-related resource usage information (blkin, blkout).

The design of the service is based on a master-slave relationship: each process manager has a co-resident master credentials server, while each file server has a co-resident slave credentials server. Instances of each of these are known as the *credentials master* and *credentials cache,* respectively.

Each credentials master maintains associations between its processes and their "keys", known as *credentials ports*. When a new process is created, a credentials port is allocated and provided to the process's emulator. All subsequent messages between the emulator and file servers contain the process's credentials port, thus allowing the file servers to map the credentials port to the relevant process information using the credentials cache. If a process has previously communicated with a particular file server then the process information is readily available, otherwise the credentials cache obtains the information from the credentials master and caches it for future use.

When a process exits or executes the `rusage()` system call, its credentials master collects resource usage information from all relevant credentials caches (i.e., the ones that are caching information for that process). In addition, the credentials caches are updated with changed information (e.g., change of process group id).

Credentials ports are implemented using Mach ports such that credentials masters hold receive rights and emulators and credentials caches hold send rights.

Interruptible File-Related System Calls

In OSF/1 AD TNC, interrupting a system call in progress (e.g., at signal delivery or `exit()` time) is more complex than in standard OSF/1 because process management and file system functionality may be implemented by different servers. This necessitates a mechanism allowing process managers to track down system calls in progress at remote file servers.

This is accomplished by registering file-related system calls in progress in both emulators and servers. When a system call message is about to be sent by an emulator, it records the destination port of the message and a unique identifier acting as a transaction id. When a system call is received by a server, the identity of the thread executing the operation is stored in a table entry keyed by the destination port and transaction id.

Then, when a process manager decides to interrupt a process's system call in progress (OSF/1 semantics are that only the first thread in the task is interrupted), an abort message is sent to the process's emulator callback thread, providing the identity of the thread whose system call should be aborted. The callback thread uses this information to determine which message should be interrupted, and then sends an interrupt message to the destination port recorded in the emulator, passing the transaction id as an argument. Upon receipt of this message, the server is able to locate the thread executing the operation (if it's still in progress) and cause it to return with EINTR (e.g., by waking it up from an interruptible wait). The transaction id is used to distinguish between multiple messages that may have been sent to the same destination port.

The case of a reply to a system call message "crossing in the mail" with an interrupt message from an emulator is handled via a retry mechanism in process management code that initiates a new interrupt sequence every one second or so until the operation is interrupted (or completes normally).

If a process calls `exit()` (or aborts abnormally), all system calls executing on behalf of the process must be interrupted, and the process's emulator mustn't be relied upon because its integrity is unknown. This is done using the credentials service (described in the previous section) which knows about all file servers that have executed system calls on behalf of a particular process. Messages are sent to the relevant file servers instructing them to abort all system calls executing on behalf of the exiting process, with a reliance on retries (as above) to handle race conditions.

Distributed `select()`

Both `select()` and `poll()` may operate on a set of open files that are serviced by multiple file servers. The emulator handles this distribution by initiating separate select requests on each open file port to determine if the condition has been met. Additionally, if the system call is able to block (due to a specific or indefinite timeout), then each server will be provided with an extra reply port argument so that it may reply to the emulator if and when the condition is met. Upon meeting at least one condition, or timing out, the emulator destroys the reply port and returns from the system call. When the port is destroyed, *dead name notifications* [Draves 90] are generated to all file servers with select operations pending for this particular `select()` system call, allowing them to clean up.

NFS Support

OSF/1 AD TNC allows a multicomputer to provide both NFS client and NFS server services [Sandberg 85].

Mounting an external NFS server into the OSF/1 AD TNC file name space is similar to mounting local file systems as described in the Distributed File Name Space section. The difference is that the "device" mounted on top of a directory is not a file system device but rather a "NFS device," where the *node number* for that device is a node that is known to be running a server containing NFS client code. The lower levels of the NFS client code have been modified to use the *virtual socket* interfaces (see Distributed Network Domain Sockets below), so the NFS client code does not need to be co-located with the networking server. If the directory is managed by a different server, then a remote mount is established in the same way as for local file systems. At this point, all accesses to the mounted NFS server "just work", in that system calls are automatically forwarded across the remote mount to the NFS client code.

NFS server services are provided by running NFS daemons (*nfsd*'s) (nfsd's can run on any node of the multicomputer). An nfsd receives requests from external clients and translates them into requests to the global OSF/1 AD TNC file system. It does this by encoding the node number of a file in the NFS file handle, and forwarding client requests (via Mach IPC) to the file server on that node. A thread in the file server handles the request by executing NFS server code just as if it was an nfsd. After execution, the reply is sent back to the nfsd, which then replies to the client.

Socket Enhancements

There are two classes of sockets used in OSF/1 AD TNC: Unix domain sockets and network domain sockets. OSF/1 AD TNC has enhancements to deal with each of these classes:

- *Unix Domain Sockets.* Performance criteria dictate that the data storage for connected pairs of Unix domain sockets, which are used to implement pipes and FIFOs in OSF/1, be kept on the same node as the "primary reader" process. There may be multiple processes

writing to a pipe, but typically there is only one process reading from a pipe. Additionally, keeping connected pairs of sockets on the same node reduces changes to the low-level Unix domain protocol code, which assumes all processes communicating through Unix domain sockets share the same server address space.

- *Network Domain Sockets.* There are typically several network-capable nodes in the multicomputer environment. For performance reasons, each of the network-capable nodes executes its own network protocol stacks (this can also be employed to provide some level of fault tolerance).

Virtual Sockets

To accommodate the goals of both true networking protocol families and the Unix domain protocols, a virtual socket layer has been added to OSF/1 AD TNC. A pointer to a virtual socket operation table is stored in each socket when it is created, the operation table chosen being determined by the specified domain, type, and protocol arguments. Socket related system calls have been modified to use macros such as `VSOP_SEND()`, `VSOP_BIND()`, etc., which make indirect calls through the operation table, rather than directly calling `sosend()` or `sobind()`. The virtual socket operation functions themselves hide the multicomputer environment from the socket manipulation code, performing function shipping and coordinating socket activity among the socket-using nodes and network-capable nodes.

Distributed Unix Domain Sockets/Pipes

When a Unix domain socket is initially created, its file server node is the execution node of the process that made the creating system call. In order to have the socket serviced by a file server on the same node as the primary reader process, the socket must be able to migrate from node to node. This migration is necessary if:

- The primary reader process itself migrates to another node.
- The primary reader process exits, in which case the socket eventually migrates to the node of another reader process. (If there are no other readers, socket migration becomes superfluous, since the next write operation will result in an error, e.g., `EPIPE` or `ECONNRESET`.)

Socket migration of Unix domain sockets is performed in several phases:

- *Notification* – the migrated process sends a message on those file ports corresponding to Unix domain socket objects.
- *Preparation* – the file server waits for outstanding operations on the notified file port to complete.
- *Relocation* – port rights and copies of socket data and meta-data are sent to the file server on the new execution node.
- *Arrival* – information sent in the relocation phase is used to reconstruct the socket object on the new node.
- *Clean-up* – data structures and resources on the old node are freed; the file server on the new node takes over responsibility for the socket object.

Although the details of each phase vary depending on the particular type of socket Unix domain socket based object involved, this schema is generally applicable to all of the types – pipe, FIFO, stream, or datagram socket. As the simplest type of socket based object, pipes closely follow the general scenario outlined above, and so for them no further elaboration is presented.

Binding Unix Domain Sockets

In the Unix domain, a socket address corresponds to a name in the file system name space. A socket and path name are presented to the `bind()` system call, which creates a vnode and associates it with the socket. In standard OSF/1, this association is just a pointer to the socket structure in system virtual address space. In OSF/1 AD TNC, the socket and vnode structures may reside on different nodes of the multicomputer, so a Mach port must be used to tie them together. This port is called the socket's *bind port.* When a `connect()` system call issues a `PRU_CONNECT` user request to the Unix domain protocol code, a bind port is obtained using `namei()`. The bind port is then passed to the `VSOP_VSOCKCONNECT()` virtual socket operation to complete the connection in a protocol-specific manner.

Unix Stream Sockets

When establishing a connection with Unix domain stream sockets, the bind port is used as an RPC handle to relocate the connecting socket and its file port to the node where the bound socket resides. The stream socket arrival routine uses pre-existing protocol code to complete the connection, and a new socket file descriptor is returned to an application program blocked in the `accept()` system call. From then on the resulting pair of connected sockets behaves just like a (bidirectional) pipe, and in particular its relocation behavior is the same as a pipe's.

Unix Datagram Sockets

Communication between Unix domain datagram sockets in standard OSF/1 is accomplished by temporarily making a connection between the sockets using pointers in server virtual address space. In general these temporary connections are very short-lived, usually only lasting for the transfer of a single datagram. In a

multicomputer environment, sockets often reside on different nodes. Given the short duration of datagram socket temporary connections, relocating a connecting socket to a bound socket on a remote node for the transfer of so little data would be inefficient.

As with Unix stream sockets, a bind port is used to establish datagram socket connections. However, datagram sockets use the bind port to send actual user data rather than to move the communicating sockets to the same node. Short term, single datagram connections are made by the `sendto()` system call, which discards the bind port send right after the data is sent; longer term connections made with the `connect()` system call keep the send right with the connecting socket until the socket is closed or the connection is explicitly broken by connecting to a different bound socket. Unix datagram sockets follow migrated processes to the new execution node as soon as possible, regardless of whether the process is a reader or a writer.

FIFOs

In standard OSF/1, the FIFO implementation is based on sockets, and an in-core FIFO vnode holds a pointer to a pair of connected Unix domain stream sockets. Opening the FIFO creates a new file port and a file structure with a pointer to this vnode. As with pipes, we want the FIFO storage to follow the primary reader process when it migrates to a new node. This presents special problems for FIFOs. Many file ports may have to be relocated, not just one or two. Because file ports and their file structures reference the FIFO vnode, the vnode itself must also be relocated, but must still be accessible to `namei()` lookups using the same path name. Finally, the virtual socket interface provides little help, since the mechanism of FIFO relocation must naturally center around the vnode layer.

The implemented solution takes advantage of the distributed file system's vnode ports. Before a FIFO relocates away from the node where its original file system vnode resides, a vnode port is created for it. This *redirect port* is used for forwarding `namei()` lookups. During relocation the receive right for the redirect port moves to the new node, where a "storage node" vnode is allocated. The vnode at the file system node holds a send right for the redirect port, and hooks in `namei()` allow subsequent lookups to be forwarded to the storage node just as lookups that cross inter-node mount points are forwarded.

Although virtual socket operations are not used, the job of moving a FIFO from node to node follows the same pattern, from notification to cleanup. The TNC FIFO implementation uses hook routines similar to virtual socket operations whenever FIFOs are opened or closed, and to receive notification messages and trigger relocation.

Distributed Network Domain Sockets

In a multicomputer, the network protocol processing is distributed among the nodes designated as network servers. A network server may handle one or more protocol stacks for one or more network interfaces. Indeed, network interfaces are not required to be on the same node as network servers. However, in typical configurations, each node with a network interface runs a network server to handle the protocol stacks for that network interface.

The multiple instances of network servers are hidden from the user by the virtual socket layer. When a network server is configured (by configuring an interface), the virtual socket layer notifies a central database, called the *clearinghouse*, with its node number, protocol address (including family), and network server port. The clearinghouse is responsible for:

- Insuring that two interfaces are not configured with the same address.
- Matching a request from the virtual socket layer for a network server with the best network server available.

Creating Network Sockets

When an application makes a `socket()` system call, a virtual socket is created on the applications node. This socket is referred to as the primary virtual socket.

Binding Network Sockets

When an application makes a `bind()` system call, the clearinghouse is called by the virtual socket layer to determine which network servers to use. If the address is a wild-card address (`INADDR_ANY` or multicast), a list of servers is returned, and the virtual socket is marked as bound to a network server (only wildcard address requests can return more than one network server). A secondary virtual socket is created on each of the network servers returned by the clearing house. If a primary virtual socket and a secondary virtual socket exist on the same node, a single virtual socket is used which functions as both the primary and secondary virtual socket.

Wildcard addresses that request a routing endpoint return only the local node's network server. This is to insure that each network server can run the routing protocols necessary to maintain routing tables. If the address is a single endpoint and the clearinghouse lookup is successful, the virtual socket is marked as bound to a network server. After a socket is bound to a network server, the clearinghouse is not used.

As an optimization, if the bound socket has only one secondary socket associated with it, the virtual socket layer migrates the primary virtual socket and its associated file structure to the node where the secondary virtual socket resides, and the primary and secondary virtual sockets are collapsed into a single virtual socket. This reduces the number of servers which interact when servicing a system call operating on that particular socket.

Connecting Network Sockets

When an application issues the `connect()` system call, and the virtual socket is not bound to a network server by a previous `bind()` system call, the virtual socket layer tries to find the best network server to use for this connection. First the virtual server queries the clearinghouse for the file server of the endpoint the virtual socket layer wishes to connect to; if this fails, the virtual socket layer queries the clearinghouse to get the "nearest" server.

Associating Primary Virtual Sockets with Network Servers

When the virtual socket layer needs to associate a virtual socket with a network server, usually from a `bind()`, `connect()`, or `sendto()` system call, the virtual socket layer queries the clearinghouse for the Mach port of the network server it should use. The clearinghouse contains a set of protocol specific routines based on the family, type and protocol of the socket which match the requested address with a network server. The clearinghouse protocol-dependent match routine is called and returns true if the supplied address matches one or more of the addresses registered in the clearinghouse. From this, a list of network server ports is created.

If no network servers are available to match the request, and the system call does not need an explicit address match (as in the case of `connect()` or `sendto()`), the virtual socket layer calls the clearinghouse with a *getnearestserver* request. The clearinghouse employs an implementation-dependent policy to select the network server. A policy function is applied the list of network server / address tuples. Possible policies are to do nothing, return the server for the address with the best (longest) match, return the list in a random order, query each network server to find the best route, etc. The current policy is one which searches the clearinghouse for a connected subnet; if that fails, it sequentially cycles through all the available network servers. An advantage of the current policy is that if two devices reside on the same network, they can be configured on separate network servers. In this configuration, unlike most SMP or uniprocessor implementations, load balancing can be achieved not only on inbound traffic, but on outbound traffic as well, since each connection request will obtain a different server.

Communication between Network Servers

While the user sees the network servers as a single instance of a protocol stack, each network server sees other network servers as if they were connected over a network. This allows protocol stacks to transmit routing information using the protocols' native routing protocols. The *mi* interface driver is used to create this internal network. The *mi* interface is a pseudo interface driver which uses Mach IPC messages to transmit and receive protocol packets among the network servers. The first incarnation of the *mi* driver was a simple point to point driver. However, due to scaling issues, a multipoint driver was developed which uses a central node for replaying broadcast and multicast packets. The clearinghouse provides proxy ARP services for the *mi* interface.

The chief advantage of using native protocols for propagating routing information is the ease of adding new protocols as they are developed – few server modifications are required.

Process Management Enhancements

In OSF/1 AD TNC, process management is distributed across all the nodes of the multicomputer. Typically, processes running on a specific node have their process management system calls serviced on the same node. It is desirable to perform this distribution without major impact on the standard OSF/1 source code. In order to do this, the Virtual Process (or *vproc*) layer was added. By analogy, the *vproc* layer does for processes what *vnode* layer does for the files. The result is location transparency at the system call interface.

A set of new remote processing primitives has been added to enable specialized user processes to exploit multicomputer resources. Through these primitives, processes can `fork()`, `exec()`, or `migrate()` onto another node. Furthermore, TNC adds a new signal, `SIGMIGRATE`, that causes a receiving process to move to another node.

In addition, OSF/1 AD TNC provides a load leveling daemon. This daemon makes extensive use of the `SIGMIGRATE` signal to distribute the workload across all the nodes of the multicomputer.

Virtual Processes

Process-related operations are requested through the *vproc* layer [Walker 93]. Only the vproc layer is aware of the physical location of – and POSIX-defined relationships between – processes. The vproc layer directs each physical process operation to the node where a process is located. The process management server on that node is called upon to complete the operation.

For local processes, the code below the vproc layer manipulates the physical process directly. For remote processes, the code below the vproc layer invokes client routines which communicate via RPCs with the process management server on the remote process's execution node to access the physical process on behalf of the client.

Process Identification

A key element of process transparency provided by the vproc architecture is a system-wide process identifier space. Subject to the limitation of 32 bit process identifiers: the high order bits are the node number of the node which created the process; the low order bits are generated uniquely by the creation node. Using this scheme each node can generate PIDs without consulting any other node and yet each PID in the system is unique.

The traditional process 1 (`init`) is treated specially. A single `init` exists for the entire multicomputer system; it is located on the root file server node, but `init` is allocated PID 1 irrespective of which node this may be.

Vproc Structure

The vproc structure is very simple. It is all that is visible of a process to base server code, and the fields in this structure can be relied upon, even if the process is not executing locally nor even known about locally. Details of physical process location, physical process attributes and process relationships are below the vproc abstraction layer and available only through vproc operations. The following is the vproc structure:

```
struct vproc {
    long vp_magic;
    pid_t vp_pid;
    unsigned long vp_ref_cnt;
    struct vproc_ops *vp_ops;
    caddr_t vp_data;
    vp_lock_t vp_ref_cnt_lock;
    struct pid_hash vp_pid_hash;
}
```

- `vp_magic` is a magic number used to ensure that a `struct vproc` is being accessed.
- `vp_pid` is the standard process identifier.
- `vp_ref_cnt` is a local reference count to this data structure. When the count reaches zero, the vproc is freed
- `vp_vops` is a pointer to a table of operations applicable to the vproc.
- `vp_data` is pointer to private data, much like one sees in a `struct vnode`.
- `vp_ref_cnt_lock` is a lock for multiprocessor locking of the `vp_ref_cnt`.
- `vp_pid_hash` is used to provide fast access to a vproc for a given PID.

Vproc Operations

Vproc operations hide the distributed implementation of processes and inter-process relationships and are necessary if:

- the operation is to be performed on a process other than the current process; or
- the operation changes or affects inter-process relationships; or
- the operation may change the status of a process in a way which may be of interest to other processes.

In OSF/1 AD TNC there are 18 such operations. One example is VPOP_SIGPROC() which sends a signal to a specified virtual process; this operation is used widely throughout the server and, in particular, in support of the `kill()` system call.

In addition to the operations on individual virtual and physical processes, there are also some operations that operate on the virtual process system as a whole. These operations are known as *virtual process system operations*.

In order to shield virtual process operations from the details of the actual physical process implementation, virtual process operations interact with individual physical processes using a well-defined set of routines, known as *physical process operations*.

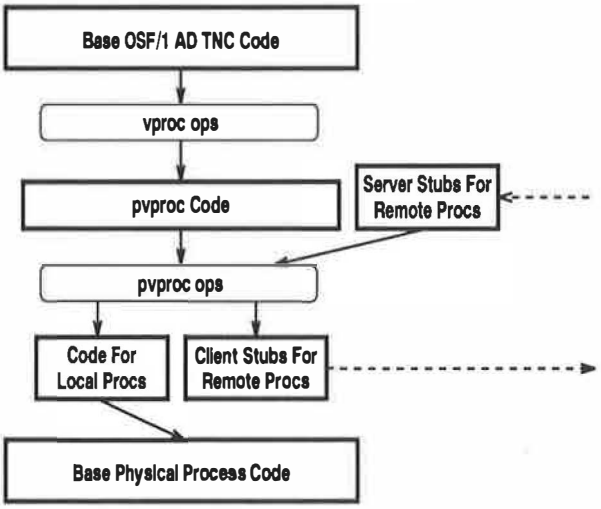


Figure 5: Distributed Process Architecture

Distributed Process Management

A vproc is independent of the execution node of the corresponding physical process and the distribution of related processes (e.g. parent). Those details are hidden within the vproc layer in a private data called `struct pvproc`. On nodes where the process is not executing there could be a vproc and pvproc but there will be no process structure. Figure 5 illustrates the distributed process architecture.

Pvproc Structure

The `pvproc` structure is pointed to by the `vp_data` field in the `struct vproc`. The pvproc structure in turn points to the physical process structure, `struct proc`. Information about process relationships (parent-sibling-child, process group and session membership) resides in the pvproc structure and not the proc structure, which is not referenced or modified directly by the TNC distributed processing code.

Pvproc Operations

Pvproc operations hide the process execution location. Each pvproc operation is executed against a specific process. A pvproc operation may or may not be called on the node where the specified process is executing. However, the operations vector for the pvproc either directs execution to physical process code (*pproc operation*) to act on a locally running process or to a routine called the remote pvproc operations client stub.

The remote pvproc client stub first determines the process's execution node. This may have to be obtained from the process's origin node. The operation is then packaged as a RPC to the execution node. OSF/1 AD TNC associates a Mach port with each vproc, and it is to this vproc port that RPCs are directed.

On the server side of the RPC, a check is made to ensure that the target process is indeed executing locally (it may have moved on). If not, an error is returned and the client code retries by redetermining where the process is executing. If the process is local, the appropriate pvproc operation is invoked, with the return value being sent back to the client.

Private Virtual Process System Operations

This form of operation is directed to be performed upon a specific node rather than upon a specific virtual process. The operations vector for *pvps operations* directs execution to act on the local node or via an RPC to the requested remote node. Such operations are principally internal to the vproc layer; for example, to obtain the current execution node of a particular process from the origin node of that process (determined from its PID).

Remote Processing Primitives

The following additional system calls are provided by OSF/1 AD TNC:

`rfork()`

Taking an additional node parameter over the conventional `fork()` system call, `rfork()` forks a child copy of the caller onto a remote node. As usual, the parent is returned the PID of its new child, but note that this will reflect the number of the node on which the child has been forked.

In all respects, the parent and child behave as if they are co-located: the child inherits its parent's process group id (PGID), session id (SID), text and data segments, open files etc. The parent can wait on its remote child in the usual way.

Remote forking exploits the Mach 3.0 `norma_task_create()` primitive to create the physical child task on the remote node inheriting the memory of the parent. The vproc layer manages internode process relationships by adding a vproc for the new child process in the parent-sibling-child list on the parent node. This child vproc provides a reference to the execution node of the child process. Conversely, a vproc for the parent process on the execution node of the child provides a link back to the execution node of the parent. In a similar manner, process group and session relationship information is maintained for the new child process.

`rforkmulti()`

This form of remote `fork()` is provided to create multiple children at once on a specified set of remote nodes as a more efficient alternative to a series of `rfork()`'s. `rforkmulti()` takes an input array parameter specifying the nodes on which remote forks are to be performed, and returns two arrays indicating the PIDs of the children and status values from each node.

Efficiency is obtained using a binary tree algorithm with the list of nodes being successively divided in half. On each remote node visited, the first element in the list is the node's own node number. Before performing the fork on the visited node, the remaining list is again divided into two parts and `rforkmulti()` is called recursively and asynchronously for each part. Completion of the two asynchronous parts is awaited after the local fork is complete.

`migrate()`

This system call relocates the calling process to the remote node specified by a single parameter. Upon successful return, the caller will be executing on the new node but in all other respects the process is unaltered: it has the same PID, PGID and SID, text and data, open files etc.

Process migration exploits the Mach 3.0 primitive `norma_task_create()` to fashion a new user task on the remote node from the image of the former user task. Process state information about the physical process is transferred to the remote node so that the process is replicated exactly in the process management server on the new node before the process is deleted from the old node.

Process relationship information is also transferred to the new node: If the process is a parent, the list of vprocs representing its child processes is transferred; if a process group leader, the list of vprocs representing the group member processes; and, if a session leader, the list of

vprocs representing the process group leaders in the session. Note that a vproc will be deleted from the old node only when no more references to it exist there.

`rexecve()`

Taking an additional node parameter over the conventional `execve()` system call, `rexecve()` executes a new program image on a remote node. Note that the PID is retained when the process moves to the new node.

A remote exec is performed as a special case of a migration followed by an `execve()`. Process relationship information is handled exactly like a migration, but less physical process state information needs to be transferred, since much of this is reset upon executing the new process image.

`kill3()`

This system call is a superset of the usual `kill()` system call. Taking an additional argument, `kill3()` sends a signal and the associated integer argument to a specified process. This semantic is of relevance only to the OSF/1 AD TNC-specific signal `SIGMIGRATE`, the default handling of which is to migrate the receiving process to the node indicated by the associated argument. Hence, `kill3()` may be used to migrate any process regardless of it being aware of the fact that migration is occurring; the load leveler employs this mechanism to migrate processes away from overloaded nodes.

Load Leveling

The function of the load leveler is to balance the load among the multicomputer nodes. In OSF/1 AD TNC, load leveling is accomplished by periodically migrating processes from overloaded nodes to underloaded nodes.

There are two aspects to load leveling:

- Exchanging load information between nodes, in order to determine overloaded and underloaded nodes.
- Migrating processes from overloaded nodes to underloaded nodes.

The OSF/1 AD TNC load leveler comprises daemon processes that run on every node of the system. Each daemon process is responsible for both exchanging load information between nodes and for determining the processes to be migrated from overloaded nodes to underloaded nodes.

Load Information Exchange

The ideal method of maintaining the load information in a multicomputer would be to keep the current load information about all nodes in the multicomputer. In this way, the process migration policy can be very accurate. Unfortunately, maintaining global load information in a large multicomputer configuration can be very expensive. Hence, a simpler, less expensive algorithm was required. The load information exchange algorithm for OSF/1 AD TNC was derived from the algorithm used by the MOS multicomputer [Barak 85b]. The load leveler daemon process for each node maintains a fixed-size load vector that is exchanged between nodes (the size of the vector is a tunable parameter). This load vector contains load information about an arbitrary (random) subset of nodes in the multicomputer.

Periodically, the load leveler daemon on each node:

- Updates its own load value, putting the load value into the first element of the load vector. The load value is a (tunable) weighted average of the node's 5-second, 30-second, and 1-minute load averages.
- Sends the first half of its load vector to the load leveler daemon process on a random other node.

When the load leveler daemon process on a node receives a portion of a load vector from another node, it discards the latter half of its current load vector and literally "shuffles in" the received portion of the load vector with the first half of its current load vector.

This load information exchange algorithm has been shown to be efficient [Barak 85b], and (with carefully tuned parameters) allows the process migration policy to perform its work accurately.

Process Migration Policy

At periodic intervals, the load leveler daemon on each node calculates the load average of all the load values contained in the load vector. It then uses this average to determine whether the local node is overloaded and whether any other nodes specified in the load vector are underloaded. The interval is a tunable parameter, along with the values used to determine whether nodes are overloaded or underloaded. If the local node is overloaded and at least one other node specified in the local load vector is underloaded, the local load leveler daemon will attempt to migrate processes to underloaded nodes.

The underloaded node that is to receive a specific extra process is probabilistically selected from the list of underloaded nodes (while any underloaded node may be selected, the more underloaded a node is, the higher the probability that it will be selected). The daemon migrates the process by invoking the `kill3()` system call, which sends a `SIGMIGRATE` signal, along with the destination node number, to the process. Processes are selected for migration until the local node is no longer overloaded, or until no underloaded nodes remain.

Booting and Configuration

OSF/1 AD TNC servers configure themselves using *boot config variables* provided to each node when the system boots. These variables are tailored specifically to each node, telling servers running on the node:

- whether to export paging services (so that a file server may provide paging storage for processes and servers on diskless nodes)
- whether to import paging services, and if so from what node
- what node is running the file server managing the root
- what node has the root file system device attached
- what nodes are network servers
- a list of all nodes in the multicomputer
- a mapping of device node numbers to file server node numbers (so that a file server and the devices it's managing need not be co-located)
- miscellaneous information such as the file name of the emulator image, debug options, etc.

The system administrator maintains a *configuration file* with boot config variables. They are loaded into the kernel at boot time (see below) and made available to user-space via the `host_get_boot_info()` kernel interface.

The initial loading of boot config variables into the kernel is inherently machine-dependent. For instance, Intel's Paragon XP/S Supercomputers use a machine-dependent mechanism for loading the boot config variables, as well as kernel, server, and emulator images. Another environment consists of a cluster of 386 clones for debugging (simulating a multicomputer), and uses a special server on the Ethernet to respond to requests from booting kernels for their boot config variables.

Given a node number, locating the corresponding file server relies on a special server, known as the *local node nameserver*. An instance of the local node nameserver runs on each node providing file service, and is registered with a *well-known* name in its local kernel. A file server, in turn, is registered with its local node nameserver. Since a server running on one node is able to look up the local node nameserver running on any other node, the system is thus able to map from a node number to a port for the node's file server. Caching is done to expedite subsequent attempts to locate the same file server.

Related Work

The architecture of an operating system designed to run on a multicomputer system is very similar to the architecture of an operating system designed to run on a networked cluster of systems (generally referred to as a distributed operating system). Indeed, OSF/1 AD TNC was originally developed upon (and still runs upon) a network of 386 clones.

There have been several distributed Unix projects in the past. Several of these projects are surveyed in this section.

The Locus Distributed Operating System [Walker 83] [Popek 85], commercialized by IBM as TCF (Transparent Computing Facility) [Walker 89], was the first commercially available system to provide full single system image in a distributed environment. TCF is a modified version of Unix System V.2, and is only available from IBM. TCF scaled only to a few dozen nodes.

IBM and Locus also have designed a distributed operating system that was based on extensions to DCE [Walker 92]. This system utilized many of the concepts developed in TCF. This system was designed to scale to hundreds or thousands of nodes, but is not commercially available. The process management enhancements of OSF/1 AD TNC are based on the concepts developed in this system and in TCF.

Sprite [Ousterhout 88] [Douglis 90] is a research system of the University of California at Berkeley designed for a network of homogeneous workstations. Sprite provides a single file system image throughout the workstations in the cluster (using a custom distributed file system) and provides load leveling by use of process migration [Douglis 87].

Amœba [Tanenbaum 91] [Douglis 90] is a research system at the Vrije University of Amsterdam. It is designed for distributed computing and highly parallel applications. It provides only partial Unix compatibility.

MOSIX [Barak 85a] is a research system at the Hebrew University of Jerusalem. It is a Unix-derived distributed operating system that provides load leveling by use of process migration. The OSF/1 AD TNC load leveling algorithms utilize concepts developed in MOSIX [Barak 85b]. MOSIX does not provide a true file location transparency, and does no remote file caching.

Chorus [Armand 89] [Guillemont 91] [Batlivala 92] provides a microkernel and servers that support Unix and distributed processing. However, Chorus does not as yet support single system image across a cluster of nodes, nor does it support load leveling.

Implementation Status

OSF/1 AD TNC is the operating system for the Intel Paragon XP/S Supercomputer [Intel 92]. Intel Paragon XP/S Supercomputers systems with more than 512 nodes have been shipped.

Additionally, OSF/1 AD TNC runs on clusters of 386 clones on an Ethernet, simulating a multicomputer.

Future Enhancements

Future enhancements to OSF/1 AD TNC will come in two areas: performance and availability.

Performance

While OSF/1 AD TNC is fully functional, it has not yet undergone rigorous performance analysis and tuning. Detailed analyses must be undertaken to discover and remove performance problems.

Specific areas that may require enhancement are:

- The addition of a prefix name cache to the emulator, to reduce the number of server-to-server boundary crossings during pathname resolution.
- The addition of file system replication, to allow file server requests for a single file system (such as the root file system) to be load leveled across several nodes.

Availability

The current reconfiguration characteristics of OSF/1 AD TNC are quite simple: nodes cannot enter or leave the configuration without rebooting the whole system. This is entirely a software constraint.

It is desirable to be able to perform reconfiguration without rebooting the system; whether to add a node to the configuration, or to perform a planned removal of a node from the configuration, or to allow for the unplanned removal of a node from the configuration (due to hardware or software problems on the node).

Additionally, a checkpoint/restart capability is being added for availability.

These availability characteristics are especially desirable on larger configurations.

Conclusion

This paper has described the features of OSF/1 AD TNC, a version of OSF/1 suitable for massively parallel multicomputers. OSF/1 AD TNC enables users to take advantage of multicomputer hardware, while retaining full OSF/1 (and thus Unix) semantics. Scalability bottlenecks are avoided by distributing control of the file system, socket protocol stacks, and process management across multiple nodes. Programs may take advantage of the computational resources of multiple nodes via explicit use of new system calls, or may rely on automatic load leveling.

OSF/1 AD TNC is the operating system for the Intel Paragon XP/S Supercomputer, which has shipped in configurations with more than 512 nodes. Future work to OSF/1 AD TNC will include further performance analysis and tuning, enhancements for greater availability in the face of node failure, and improved reconfiguration characteristics so that nodes may enter and leave the configuration without rebooting the entire system.

Acknowledgements

We thank the folks at Intel's Supercomputer Systems Division for their contributions to OSF/1 AD TNC, including Don Cameron, Charles Johnson, Andrew Pfiffer, Paul Pierce, and Stan Smith. We also thank Joseph Barerra, Jeffrey Heller, Alan Langerman, and Steven Sears for their work improving the Mach kernel.

References

[Armand 89] Armand, F., Gien, M., Hermann, F., Rozier, M. Distributing UNIX Brings it Back to its Original Virtues. In *Proceedings of Workshop on Experiences with Building Distributed (and Multiprocessor) Systems*, October 1989.

[Barak 85a] Barak, A., Litman, A. MOS: A Multicomputer Distributed Operating System, In *Software – Practice and Experience*, Vol. 15(8), August 1985.

[Barak 85b] Barak, A., Shiloh, A. A Distributed Load-balancing Policy for a Multicomputer. In *Software – Practice and Experience*, Vol. 15(9), September 1985.

[Barrera 91] Barrera, J. A Fast Mach Network IPC Implementation. In *Proceedings of the USENIX Mach Symposium*, Nov. 1991.

[Barrera 93] Barrera, J. Copying, Caching, and Sharing in a Distributed Virtual Memory System. Ph.D. Dissertation, Carnegie-Mellon University, to appear 1993.

[Batlivala 92] Batlivala, N., Gleeson, B., Hamrick, J., Lurndal, S., Price, D., Soddy, J., Abrossimov, V. Experience with SVR4 over Chorus. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Architectures*, April 1992.

[Bershad 90] Bershad, B., Anderson, T., Lazowska, E. Lightweight Remote Procedure Call. *ACM Trans. on Computer Systems*, 8, 1, February 1990.

[Black 88] Black, D., Golub, D., Hauth, K., Tevanian, A., Sanzi, R. The Mach Exception Handling Facility. In *Proceedings of the ACM Workshop on Parallel and Distributed Debugging*, May 1988.

[Bricker 91] Bricker. A., Gien, M., et al. Architectural issues in microkern-based operating systems: the CHORUS experience. In *Computer Communications, Special Issue: Platforms for Distributed Applications*, July/August 1991.

[Cheriton 89] Cheriton, D., Mann, T. Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance. *ACM Trans. on Computer Systems*, 7, 2, May 1989).

[Dean 92] Dean, R., Armand, F. Data Movement in Kernelized Systems. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Architectures*, April 1992.

[Douglis 87] Douglis, F., Ousterhout, J. Process Migration in the Sprite Operating System. In *Seventh International Conference on Distributed Computing Systems*, September 1987.

[Douglis 90] Douglis, F., Kaashoek, M. F., Tanenbaum, A. S. A Comparison of Two Distributed Systems: Amoeba and Sprite. In *Vrije University Report IR-230*, December 1990.

[Draves 90] Draves, R. A Revised IPC Interface. In *Proceedings of the USENIX Mach Symposium*, October 1990.

[Forin 91] Forin, A., Golub, D., and Bershad, B. An I/O System for Mach 3.0. In *Proceedings of the Second USENIX Mach Symposium*, November 1991.

[Golub 90] Golub, D., Dean, R., Forin, A., Rashid, R. Unix as an Application Program. In *Proceedings of the Summer 1990 USENIX Conference*.

[Guillemont 91] Guillemont, M., Lipkis, J., Orr, D., Rozier, M. A Second-Generation Micro-Kernel Based UNIX; Lessons in Performance and Compatibility. In *Proceedings of the Winter 1991 USENIX Conference.*

[IEEE 90] Portable Operating Systems Interface (POSIX) - Part 1: System Application Program Interface. IEEE Std 1003.1, 1990. ISO/IEC 9945-1.

[Intel 92] Intel Paragon XP/S Supercomputer Spec Sheet. Intel Corporation, 1992.

[Julin 91] Julin, D., Chew, J., Stevenson, M., Guedes, P., Neves, P., Roy, P. Generalized Emulation Services for Mach 3.0 – Overview, Experiences, and Current Status. In *Proceedings of the USENIX Mach Symposium*, November 1991.

[Kleiman 86] Kleiman, S. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *Proceedings of the Summer 1986 USENIX Conference.*

[Loepere 92] Loepere, K. (editor). *Mach 3 Kernel Interfaces*. Open Software Foundation and Carnegie Mellon University.

[OReilly 91] Guide to OSF/1: A Technical Synopsis. O'Reilly & Associates, Inc., 1991.

[OSF 90] The Design of the OSF/1 Operating System. Open Software Foundation, 1990.

[Ousterhout 88] Ousterhout, J., Cherenson, A., Douglis, F., Nelson, M., Welch, B. The Sprite Network Operating System. In *Computer*, February 1988.

[Paciorek 92a] Paciorek, N., Teller, M., Black, D., Roy, P. Design Specification for an OSF/1 Based Distributed File Server on Mach 3.0. Open Software Foundation Research Institute, 1992.

[Paciorek 92b] Paciorek, N., Teller, M. An Object Oriented, File System Independent, Distributed File Server. In *Proceedings of the USENIX File System Workshop,* May 1992.

[Popek 85] Popek, G., Walker, B. *The LOCUS Distributed System Architecture*, MIT Press, 1985.

[Sandberg 85] Sandberg, R., Goldberg, D., Kleiman, S., Lyon, B. Design and Implementation of the SUN Network Filesystem. In *Proceedings of the Summer 1985 USENIX Conference.*

[Sansom 86] Sansom, R., Julin, D., Rashid, R. Extending a Capability Based System into a Networked Environment. In *Proceedings of the ACM SIGCOMM 86 Symposium on Communications Architectures and Protocols*, August 1986.

[Tanenbaum 91] Tanenbaum, A. S., Kaashoek, M. F., van Renesse, R., Bal, H. E. The Ameoba distributed operating system – a special report. In *Computer Communications, Special Issue: Platforms for Distributed Applications*, July/August 1991.

[Walker 83] Walker, B., Popek, G., English, B., Kline, C., Thiel, G. The LOCUS Distributed Operating System. In *Proceedings of the Ninth Symposium on Operating Systems Principles*, October, 1983.

[Walker 89] Walker, B., Popek, J. Distributed UNIX Transparency: Goals, Benefits, and the TCF Example. In *Winter 1989 Uniforum Conference Proceedings.*

[Walker 92] Walker, B., Lilienkamp, J., Hopfield, J., Zajcew, R., Thiel, G., Matthews, R., Mott, J., Lawlor, F. Extending DCE to Transparent Processing Clusters. In *Winter 1992 Uniforum Conference Proceedings.*

[Walker 93] Walker, B., Zajcew, R., Thiel, G. Vprocs: A Process Abstraction to Enable Single System Image Distributed Computing. Submitted for publication.

[Welch 86] Welch, B., Ousterhout, J. Prefix Tables: A Simple Mechanism for Locating Files in a Distributed Filesystem. *Proceedings of the 6th ICDCS*, May 1986.

[Young 87] Young, M., Tevanian, A., Rashid, R., Golub, D., Eppinger, J., Chew, J., Bolosky, W., Black, D., Baron, R. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of the Eleventh ACM*

Symposium on Operating Systems Principles, November 1987.

[Young 89] Young, M. Exporting a User Interface to Memory Management from a Communication-Oriented Operating System. Ph.D. Dissertation, Carnegie-Mellon University, November 1989. CMU-CS-89-202.

Author Information

Roman Zajcew is a Senior Scientist at Locus Computing Corporation in San Diego, where he is working in the area of distributed computing. An ex-Canadian, he is exploring the United States by attending meetings of assorted industry working groups and standards bodies. He graduated from the University of Manitoba with M. S. and B. S. degrees in Computer Science. Prior to Locus, he worked on SMP System V for Encore Computer Corp., on an object-oriented operating system for Syte, on kernel internals for NCR, and on telephony for Bell Northern Research. His e-mail address is roman@locus.com.

Paul Roy is a Senior Research Engineer at the OSF Cambridge Research Institute, where he is working on microkernel-based operating systems and new file caching architectures for OSF/1. Prior to OSF, he was in the Operating Systems Group at Apollo Computer, and was a member of the Distributed Systems Group at Stanford University. Paul received a M.S. degree in Electrical Engineering from Stanford University in 1985 and a B.S. degree in Computer Engineering from the University of Massachusetts in 1983. His e-mail address is roy@osf.org.

David Black is a Senior Research Fellow at the OSF Cambridge Research Institute, where his current work involves a joint research project with Carnegie Mellon University and other collaborators on micro-kernel based operating system environments. He has also contributed to the design and implementation of portions of the OSF/1 operating system. Dr. Black was one of the key developers of the Mach operating system at CMU, and received the Ph.D. degree in Computer Science in 1990. He also holds an M.S. in Computer Science from CMU, and a B.A. and M.A. in Mathematics and a B.S.E. in Computer Science and Engineering from the University of Pennsylvania. Dr. Black's research interests include microkernel technology, real-time, parallel, and distributed computing. His e-mail address is dlb@osf.og.

Chris Peak has a degree in Mathematics from Cambridge University, England. He is currently a Consulting Member of Technical Staff with Locus Computing Corporation in San Diego. Before moving to the US, he worked for British systems house Logica. He is interested in getting things right. His e-mail address is chrisp@locus.com.

Paulo Guedes received a B.S in Electrical Engineering and a M.S. in Computer Science from the Technical University of Lisbon (IST) and is currently pursuing his Ph.D. in distributed object systems at IST. He was on sabbatical at the OSF Cambridge Research Institute for two and a half years. His interests are in operating systems and object-oriented programming. His e-mail address is pjg@inesc.inesc.pt.

Bradford Kemp is a Scientist at Locus Computing Corporation. He has been active in the development and standardization of of communication protocols at both the international and national levels. At Locus he designed and developed the Virtual Socket Layer. He attended Brown University until 1981 in search of a B.S. in Computer Science. His e-mail address is bhk@locus.com.

John LoVerso is a Senior Research Engineer at the OSF Cambridge Research Institute, where he is working on operating systems for multicomputers. Prior to that he worked on high speed FDDI controllers at Concurrent Computer Corporation, and on the Annex terminal server at Xylogics, Inc. and Encore Computer Corporation. He received a B.S. degree in Computer Science from the State University of New York at Buffalo in 1985, and has occasionally worked on a Masters degree ever since. He can be reached as loverso@osf.org.

Michael Leibensperger is a Senior Member of Technical Staff at the Boston office of Locus Computing Corporation. Before joining Locus he was an unemployed drifter in South Asia; before that he worked in the operating systems and custom products groups at MASSCOMP. He did time at Carnegie-Mellon University. His e-mail address is mjl@locus.com.

Michael Barnett is a Principal Member of Technical Staff at Locus Computing Corporation in San Diego. He graduated from the University of Minnesota and obtained the following three degrees: Bachelor of Electrical Engineering, Bachelor of Physics, Bachelor of Astrophysics. Before joining Locus he worked on system software for Honeywell, Control Data, Lockheed (Space Shuttle Project), and Linkabit/Hughes Network Systems. His e-mail address is mbarnett@locus.com.

Faramarz Rabii has been a Senior Research Engineer at the OSF Cambridge Research Institute since Aug. 1991. Prior to that he was a Senior Software Engineer at Sequent Computer Systems, where he worked on the Dynix Multiprocessor operating system. He also worked at Locus Computing Co. from 1987 to 1990, where he worked on the TCF distributed heterogeneous operating system. He received S.B.'s in Computer Science and Chemistry from MIT in 1981 and 1984, respectively, and an M.S. in Theoretical Chemistry from

University of Pennsylvania in 1986. His e-mail address is rabii@osf.org.

Durriya Netterwala has been a Senior Research Engineer at the OSF Cambridge Research Institute since Nov. 1991. Prior to that she was in the Operating Systems Group at Apollo/Hewlett-Packard, where she worked on the Domain/OS Operating System. She received her M.S. in Computer Science from University of Michigan, Ann Arbor in 1987 and an M.S. in Mathematics from the Indian Institute of Technology at Bombay, India. Her e-mail address is durriya@osf.org.

An Implementation of UNIX on an Object-oriented Operating System

Yousef A. Khalidi **Michael N. Nelson**

Sun Microsystems Laboratories, Inc.

Abstract

This paper describes an implementation of UNIX on top of an object-oriented operating system. UNIX is implemented without modifying the underlying mechanisms provided by the base system. The resulting system runs dynamically-linked UNIX binaries and utilizes the services provided by the object-oriented system.

1 Introduction

In this paper we describe an implementation of a UNIX system built using Spring, an experimental object-oriented operating system developed by our research group at Sun Microsystems Laboratories. The UNIX implementation presented here is a subset of SunOS 4.1 and runs most SPARC International SCD 1.1 compliant programs.

1.1 Motivation

A problem that we faced once we built our operating system kernel and a set of core system services was how to proceed with building an application base. Other new systems when faced with the same problem have either built an application base, ported an application base, or provided the ability to run binaries from another system such as UNIX. We chose to use the last approach because we wanted to start using our system to build interesting applications using the base object model provided by Spring, without having to first implement or port a window system, an editor, and a compiler. Therefore, we decided to implement a UNIX subsystem to be able to leverage the vast majority of existing UNIX applications. Moreover, building the UNIX system on Spring served as a proof of the viability of the Spring design and as a way to exercise the system.

This paper is organized as follows: in the remainder of this section we give a brief overview of the Spring system. Section 2 lists the design goals of the implementation. Section 3 gives an overview of the architecture while Section 4 describes the implementation in detail. The implementation status is presented in Section 5. A comparison to other related work is presented in Section 6. Finally, conclusions and several possible extensions to this work are listed in Section 7.

1.2 Spring Operating System

Spring is a distributed, multi-threaded, extensible operating system that is structured around the notion of *objects*. A Spring object is an abstraction that contains state and provides a set of methods to manipulate that state. The description of the object and its methods is an *interface* that is specified in an *interface definition language*. The interface is a strongly-typed contract between the implementor (*server*) and the *client* of the object.

A Spring *domain* is an *address space* with a collection of *threads*. A given domain may act as the server of some objects and the clients of other objects. The implementor and the client can be in the same domain or in a different domain. In the latter case, the representation of the object includes an unforgeable nucleus *door* (or *handle*) that identifies the server domain.

Since Spring is object-oriented it supports the notion of *interface* inheritance. Spring supports both notions of *single* and *multiple* interface inheritance. Interface inheritance is an important factor in making Spring extensible. An interface that accepts an object of type *foo* will also accept an instance of a subclass of foo. For example, the *address_space* object has a method that takes a *memory_object* and maps it in the address space. The same method will also accept *file* and *frame_buffer* objects as long as they inherit from the memory_object interface.

The Spring kernel supports basic cross domain invocations and threads, low-level machine-dependent handling, as well as basic virtual memory support for memory mapping and physical memory management. A Spring kernel does not know about other Spring kernels—all remote invocations are handled by a *network proxy* server. In addition, the virtual memory system depends on external pagers to handle storage and network coherency.

A typical Spring node runs several servers in addition to the kernel (Figure 1). These include the domain manager and the virtual memory manager; a name server; a file server (that also acts as a default system pager); a linker domain that is responsible for managing and caching dynamically linked libraries; a network proxy that handles remote invocations; and a tty server that provides basic terminal handling as well as frame-buffer and mouse support.

The Spring file system supports cache coherent files. File objects inherit from the memory_object interface and therefore can be memory mapped. The file system uses the virtual memory system to provide data caching and uses the operations provided by the virtual memory manager to keep the data coherent. It consists of two types of file servers, one that stores data on local disks and handles cache coherency for local files, and another that utilizes virtual memory to provide caching for read and write operations and to cache file attributes for remote files. The file system also acts as the system pager.

1.3 Spring Naming

One particularly important component of the Spring architecture is the Spring naming model. In this section we describe the Spring naming model, and then in Section 4 we describe how we emulate the UNIX file system naming model on top of the Spring model.

The Spring naming service allows any object to be associated with any name. A name-to-object association is called a *name binding*. Each name binding is stored in a context. A *context* is an object that contains a set of name bindings in which each name is unique. An example of a context is a UNIX file directory. An object can be bound to several different names in possibly several different contexts at the same time.

Since a context is like any other object, it can also be bound to a name in some context. By binding contexts we can create a *naming graph*—a directed graph with nodes and labelled edges where the nodes with outgoing edges are contexts. The UNIX file system is a naming graph that is frequently restricted to a tree. We can use more complex names for referring to an object in a naming graph. Given a context in some naming graph, we can use a sequence of names to refer to an object; the sequence of names defines a path in the naming graph to navigate the resolution process. Such a sequence of names is called a *compound name*. UNIX path names are an example of compound names.

Objects are retrieved from a context by invoking the *resolve* method, bound by invoking the *bind* method, and removed by invoking the *unbind* method. In addition there are methods to obtain a list of all of the bindings in a context, to create contexts, to retrieve multiple objects at once, and to get the statistics about a context.

Spring contexts provide support for the Spring security model. When an object is bound, an ACL can be given that specifies what principals are allowed what rights on an object. When an object is resolved, a set of desired rights can be specified and an object with the desired rights will be returned if the client doing the resolve is allowed the requested rights.

Each domain has a context object that implements the per-domain name space. Each per-domain name space shares a set of bindings with other domains. Thus all domains have part of their name space in common, but they can also customize their name space as appropriate. Our naming sys-

tem is based on the architecture described in [2] including the per-process view feature which is also described in the Plan-9 naming system [3].

2 Design Goals

We started this effort to provide a UNIX subsystem with several goals in mind:

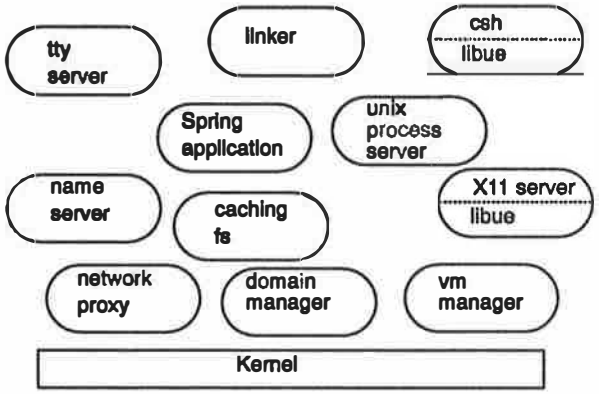
- **No modifications to Spring.** Spring was designed as an open extensible system. A major goal was to implement UNIX using existing Spring primitives and services without modifying the base system.
- **Support for dynamically linked executables.** Dynamically-linked executables that run on SunOS 4.1 should run without modifications on our system.
- **Security.** Applications must not be able to violate UNIX protections.
- **Interoperability.** Interoperability between native Spring applications and UNIX programs and libraries is a design goal. In particular, Spring applications should be able to use UNIX libraries (e.g. Xlib), Spring applications should be able to start UNIX programs, and UNIX programs should be able to exec Spring applications.
- **Performance.** Degradation in performance due to our UNIX subsystem should be minimal. The performance of applications should be a function of the underlying Spring software and hardware, and there should be a minimal performance penalty imposed by the UNIX subsystem.

Providing a complete implementation of UNIX and support for statically linked UNIX binaries were not goals of this project. We felt that we had neither the resources nor the need for such functionality. It is worth noting, however, that it is possible to provide complete UNIX support, including running statically linked binaries with our design (see Section 7).

3 Overall architecture/design overview

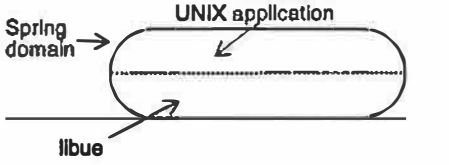
In implementing UNIX on Spring, we wanted to use the services already provided by the underlying system. Spring provides a powerful naming architecture, a distributed coherent file system, a flexible virtual memory system and support for several devices. We did not want to rewrite any of these functions. Moreover, we wanted the resulting UNIX subsystem to be "clean" and free from copyright restrictions. Therefore we did not use any pre-existing UNIX code in writing the UNIX subsystem.

Figure 1. Major system components of a Spring node



The implementation consists of two components: a library (*libue.so*) that is dynamically linked with each UNIX binary, and a set of UNIX-specific services exported via Spring objects implemented by a server domain (*UNIX process server*). The main criterion used to decide whether a certain function belongs to *libue.so* or the server is simple: as long as security is not compromised a function belongs in *libue.so*. The UNIX process server on the other hand implements functions that are not part of the Spring base system and which cannot reside in *libue.so* due to security reasons.

Figure 2. UNIX application on Spring



dynamically-linked with each UNIX application, contains:

- stubs for man (2) system calls
- a list of fd->object translations
- *unix_process* object
- a helper thread to handle signal delivery
- *libc* except for man (2) system calls

3.1 libue.so

The *libue.so* library encapsulates some of the functionality that normally resides in a monolithic UNIX kernel. In particular, it delivers signals forwarded by the UNIX process server (Section 4.3), and keeps track of the association between UNIX file descriptor numbers (fd's) and Spring objects. It also contains a *helper_thread* which is used in delivering signals and in program start-up (Section 4).

The library maintains a data structure called the *fd_table* that consists of an array indexed by fd numbers returned by the open (2) call. Each element of the array contains a pointer to an object that is a subclass of the C++ class *descriptor*. This class defines virtual methods for reading, writing, stating, selecting, asynchronous IO, and IO controls. The implementation of the *descriptor* base class provides generic implementations for these methods that are sufficient for most subclasses. Subclasses of *descriptor* can override this generic support by defining their own implementations of the appropriate virtual methods. Subclasses of descriptor are listed in Table 1.

TABLE 1. Descriptors maintained by *libue.so*

descriptor subclass	Spring object encapsulated
file_descriptor	file
tty_descriptor	tty
pipe_descriptor	unix_pipe
pty_descriptor	slave_pty
fb_descriptor	frame_buffer
io_descriptor	io.sequential_io
kbd_descriptor	keyboard
ms_descriptor	mouse
socket_descriptor	unix_socket

For each man (2) system call, we implemented a library stub. In general, there are three kinds of calls:

1. Calls that simply take as an argument an fd, parse any passed flags, and invoke a Spring service (e.g. read (2), write (2) and mmap (2)). Most of file system and virtual memory operations fall in this category.
2. Similar to (1) but eventually call out to a UNIX-specific service in the UNIX process server. Examples include pipe (2) and kill (2).
3. Calls that change the local state without calling out to any other domain. dup (2), some fcntl (2) and many signal handling calls fall into this category (main exceptions are kill (2) and killpg (2)).

We do not change *libc* or any other library. Instead when a program is exec'd (Section 4.2), *libue.so* is dynamically linked with the application image in place of *libc*. *libue.so* contains most of *libc* except that we remove the man (2) system calls from *libc* and substitute our stubs instead.

3.2 UNIX process server

TABLE 2. UNIX-specific objects

object	inherits from	example methods
unix_process	—	get_pid(), get_socket()
unix_pipe	io.sequential_io	read(), write()
unix_socket	io.sequential_io	connect(), read()
master_pty	io.sequential_io	start_output()
slave_pty	tty.tty	flush_output()

The main functions of the UNIX process server are to maintain the parent-child relationship among processes, to keep track of process and group ids, to provide sockets and pipes, and to forward signals. The objects that the server implements that are used to provide this functionality are listed in Table 2. This section describes these objects and their implementation.

The UNIX process server implements one *unix_process* object for each UNIX domain. This object is passed to each domain as part of the fork() operation (see Section 4.1). The *unix_process* object represents the identity of each UNIX process and encapsulates its process id, user id, and the resources held by the process. When a call arrives on this object, the server knows which process made the call and proceeds accordingly. For example, if the call is a *send_signal*() method, the server can decide whether or not the caller has the permission to send a signal to the destination process. Similarly, if the call allocates a socket or a tty, the server associates the allocated resource with the calling process. Methods on this object fall into four categories: methods to get/set ids of process/parent/group; methods for sending and handling signals (Section 4.5); process control methods (fork, wait and exit; Section 4.1); and methods to obtain sockets, pipes, and ptys (see below).

The UNIX process server implements one *unix_pipe* object for each UNIX pipe in the system. *libue.so* obtains

unix_pipe objects from the UNIX process server by invoking the get_pipe() method on its *unix_process* object. *Unix_pipe* inherits from *io.sequential_io* and does not add any more methods. (The Spring interface *io.sequential_io* provides methods to read and write a sequential stream). In the current implementation, data read and written to pipes pass through the UNIX process server (see Section 7 for a possible alternative implementation).

Sockets are implemented by the UNIX process server via *unix_socket* objects. There is one *unix_socket* object for each socket in the system. These objects inherit from the Spring *io.sequential_io* interface and add several socket-specific methods. Socket objects are obtained by calling the get_socket() method on the *unix_process* object. Local connections go through the UNIX process server, while remote connections go through the network proxy. The current implementation supports SOCK_STREAM and SOCK_DGRAM types in PF_UNIX and PF_INET domains. Sockets and pipes share the same underlying implementation.

Pseudo ttys are implemented with the *master_pty* and *slave_pty* objects. The *master_pty* object provides the master side of a pty. This object inherits from the Spring *io.sequential_io* interface and adds methods such as stop_output and enable_packet_mode that are required to implement the semantics of a UNIX master pty. The *slave_pty* object provides the slave side of a pty. It acts just like a tty so it inherits from the Spring interface tty.tty. Master and slave ptys are obtained by *libue.so* from the *unix_process* object methods get_master_pty() and get_slave_pty() respectively.

4 Implementation of major system components

4.1 Fork

Most of the work of forking a domain is done within *libue.so* but some help is required from the UNIX process server. Note that since our current implementation is based on SunOS 4.x, we assume a single-threaded UNIX application. We refer to this thread as the *main_thread*. Forking a UNIX domain on Spring goes through the following steps:

1. The *main_thread* which invoked the *fork* system call goes to sleep after waking up the *helper_thread*.
2. The *helper_thread* wakes up, saves the current register state of the *main_thread* in the static structure *fork_regs*, creates a new domain, and contacts the UNIX process server to inform it that this domain is forking.
3. The UNIX process server makes a copy-on-write copy of the parent domain's memory into the child domain and returns a *unix_process* object for the child.
4. The *helper_thread* packages up the file descriptors, invokes the child with these file descriptors and the child's *unix_process* object, and wakes up the *main_thread*.
5. The *main_thread* wakes up and returns from the fork system call.

The newly created child domain begins executing in the start-up code in *libue.so*. The thread that is executing in this start-up code is the child's *main_thread*. The start-up sequence for a forked child is the following:

1. The child's *main_thread* unmarshals the file descriptors and the *unix_process* object, creates the *helper_thread*, creates a *signal_handler* object (see Section 4.3) and passes it to the UNIX process server via the *unix_process* object, and does other miscellaneous initialization.
2. The *main_thread* wakes up the helper thread and then goes to sleep.
3. The *helper_thread* restores the *main_thread's* registers from the *fork_regs* structure where they were saved by the parent before its address space was copied and resumes the *main_thread*.
4. The *main_thread* wakes up and returns 0 from the fork system call.

4.2 Exec

Execing a new domain can be done entirely within *libue.so*. Our current implementation of *exec* is simple but not as efficient as possible (see Section 7 for a discussion of more efficient ways of implementing *exec*). Execing a new UNIX domain is done by creating a new domain, initializing its address space, dynamically linking the program image (more about this later), packaging up the current domain's file descriptors, and then invoking the new domain. Once the new domain is invoked, the domain

that performed the exec is destroyed. When the newly exec'd domain begins execution it merely unmarshals the file descriptors and its *unix_process* object, creates the *helper_thread* and the *signal_handler* object, registers the *signal_handler* object with the UNIX process server, and then calls the main program.

Creating a new UNIX domain during exec involves dynamically linking together the new image. In Spring there is a separate domain that performs dynamic linking. When a UNIX domain execs it dynamically links the new image by calling the dynamic linker domain which returns a set of *<memory_object, address, length>* tuples for the new image. One of these memory objects will be *libue.so* which was linked in place of *libc.so*[1]. These memory objects are then mapped into the new domain's address space at the given address for the given length. These memory objects along with memory objects for stacks and heap comprise the UNIX domain's address space.

Unfortunately it is not sufficient to merely replace *libc.so* with *libue.so*. The reason is that the standard UNIX start-up code in drt0.o that is linked in with each UNIX binary contains system call traps to dynamically link the image on UNIX. We need to replace this start-up code with Spring UNIX emulation start-up code. We do this by inserting a special *startup.so* shared library as the first dynamic library to be linked into the image; this requires special support from our dynamic linker domain[2]. This *startup.so* contains the normal Spring crt0.o and drt0.o code with some additions for UNIX emulation. The final step that this special start-up code does is call into an initialization function in *libue.so* which does things such as unmarshal the file descriptors. Thus a UNIX emulation domain is not started at the entry point given in the binary but rather at an entry point in the special *startup.so*.

Our implementation of *exec* has to be able to start native Spring domains as well as UNIX domains. In order to do this we have to know whether a program that we are starting was compiled for Spring or UNIX. We make this possible by putting a magic number right after the a.out header in each program binary that is compiled for Spring. If a program binary has this magic number then we realize that it is a Spring domain.

We start Spring domains from UNIX emulation by invoking the *start_spring_domain* method on the current process's *unix_process* object. We have to involve the UNIX process server because we need someone to deal with UNIX signals and Spring domains don't understand UNIX signals. Once the *start_spring_domain* method returns, the domain that invoked it destroys itself because it is no longer needed.

The implementation of the *start_spring_domain* method on the UNIX process server starts the Spring domain running and records the fact that the new domain is a Spring domain. If any signals are sent to the Spring domain the UNIX process server takes the default action. For example, if a SIGINT is sent, then the UNIX process server will kill the Spring domain and if SIGTSTP is sent then the UNIX process server will stop the Spring domain. Thus Spring domains can be controlled from their UNIX parent just like normal UNIX domains.

4.3 Starting a UNIX Process from Spring

The previous section discussed how we start Spring domains from UNIX domains and UNIX domains from UNIX domains, but we have not discussed how we start UNIX domains from Spring domains. In our initial implementation we had a special program called *unix_init* that could be started from Spring. This program would start a *csh* as the first UNIX program and then other UNIX programs could be started from the *csh*. However, in order to get full interoperability between UNIX programs and native Spring programs we decided that it was desirable to be able to start any UNIX program from any Spring domain.

We use the magic number described in the previous section to help us start UNIX programs from Spring domains. The standard Spring library code that is responsible for starting new domains looks at the magic number of the program that it is starting. If it doesn't have the magic number, then it is assumed to be a UNIX program. In this case the program is linked with the special *startup.so* shared library at the front. Once the program is linked it is started like any other Spring domain. Thus the only special

1. *Libc.so* is replaced by *libue.so* by merely having a symbolic link from *libc.so* to *libue.so*. Thus when the dynamic linker tries to link *libc.so* it will end up actually linking *libue.so*.

2. The only special support for UNIX emulation domains is that the Spring dynamic linker allows an extra shared library to be inserted at the front of the list of shared libraries linked in with an image. The dynamic linker itself knows nothing about UNIX emulation, it just knows how to handle an extra shared library.

support that we have in the standard Spring library for UNIX emulation is a couple of lines of code in the routine that starts new domains.

When a UNIX domain that was started from Spring begins running, the start-up code in *libue.so* discovers by looking at its environment that it was started from a Spring domain instead of a UNIX domain. Once it discovers this it performs all necessary initialization to turn this domain into a true UNIX domain. This includes doing things such as contacting the UNIX process server and informing it of the new domains existence.

4.4 File Operations

The Spring base system supports file system objects and operations that are analogous to the UNIX file system. Thus, it is easy to emulate basic file system operations such as read, write, and stat. The main complexity with emulating the file system calls are handling naming issues, selecting, and asynchronous IO.

4.4.1 Basic Operations

As we mentioned in Section 3.1 *libue.so* maintains an *fd_table* that contains one entry for each UNIX file descriptor. Each of these entries points to an object that is a subclass of *descriptor*. Entries are added to this table by UNIX system calls such as *open*, *pipe*, and *dup*. When one of the basic operations on a UNIX file descriptor such as read, write, or fstat is invoked, the appropriate method on the descriptor object pointed to by the given *fd_table* entry is invoked. For example if the read system call is made with file descriptor *fd*, the *read* method on the descriptor object pointed to by *fd_table[fd]* is invoked.

4.4.2 Naming

The Spring naming model and the UNIX file system naming model differ in two important ways. One basic difference is that whereas the UNIX file system can only name files, directories, and devices, the Spring naming system can name all types of objects. Thus in order to allow UNIX programs to live in the Spring world we have to decide if an object being resolved is of an acceptable type to UNIX. All operations except for *open* will work on any type of object that inherits from the *io.sequential_io* interface. However, *open* will currently work only on a subset of Spring objects. Currently we use a simple policy for determining if an object is acceptable to *open*:

- If the name of the object begins with "/dev" then we discern its type from its name (e.g. "/dev/mouse" refers to an object of type *mouse*) and get the desired object from the Spring name space. If we can find the corresponding Spring object, then the object is acceptable.
- Otherwise, unless the name resolves to a Spring *context* object or a Spring *file* object, the object is deemed unacceptable to a UNIX program.

A more general solution to this problem that will allow other types of objects to be accessed by UNIX programs is discussed in Section 7.

The other basic difference between UNIX and Spring naming is that Spring naming model does not support "..". The reason is that a Spring context can be bound in any number of contexts so there is no notion of a "parent" directory like there is in UNIX. In order to handle the lack of support for ".." in the Spring naming system we keep the current directory as an absolute path name. Thus all the *chdir* system call does is change the path name that is kept by *libue.so*. When we encounter a relative path name in a system call we merely append this path name to the current directory path name. If a path name has any ".." entries in it, we modify the path name to remove these entries. For example if the current directory were "/foo/bar" and we were given the relative name "../lah" we would produce the absolute path name "/foo/lah".

The disadvantage of keeping the current directory as an absolute path name is that we don't have the same semantics as UNIX when someone changes the current working directory through a symbolic link. In UNIX ".." will go up towards the real root on the current file system, but in our world ".." will just take the last component off the current directory string. In practice we haven't found this to be a problem.

4.4.3 Select

The *select* system call is implemented entirely in *libue.so* using Spring threads. We use threads for two purposes: waiting for a descriptor to become ready and for time-outs. When a user invokes *select*, we poll all the descriptors on which they are selecting to see if any are ready. If none are ready then we create a thread for each descriptor and have this thread wait until the descriptor's Spring object is ready. When a thread returns from the wait it marks the descriptor as ready. If a user invokes *select* repeatedly, we

only start threads on those descriptors for which there is not already a thread waiting.

We also use a thread for time-outs. We have one time-out thread for each domain. Before *select* goes to sleep waiting for a descriptor to become ready the time-out thread is made to go to sleep for the given time-out value. If it wakes up, it wakes up the sleeping thread that is doing the *select*.

4.4.4 Asynchronous IO

Asynchronous IO is implemented using callback objects. When a user puts a descriptor in asynchronous IO mode a callback object that is implemented by *libue.so* is installed with the implementor of the descriptor's Spring object. When the object becomes ready, the object manager invokes the callback object, *libue.so* handles the callback and sends a SIGIO signal to the current domain.

4.5 Signals

There are two types of signal system calls: those that send signals (e.g. *kill*) and those that modify the process's signal state (e.g. *sigsetmask*). In our implementation of UNIX we are able to handle the second type of system call locally without crossing into a different domain. Thus most signal system calls which would have required a kernel trap in standard UNIX are merely procedure calls in Spring.

The signal calls that send signals obviously cannot be done without leaving the current domain. There are two parts to the signal mechanism: requesting that a signal be sent to a process and handling the signal request at the signalled process. Requesting that a signal be sent involves the UNIX process server and handling the signal involves the signalled domain.

4.5.1 At the UNIX process server

The *kill* call invokes the process' *unix_process* object requesting that a signal be delivered to a particular process. The UNIX process server checks that the sending process can signal the destination process and then forwards the signal request to the destination process by invoking the *deliver_signal* method on the *signal_handler* object of the recipient.

The UNIX process server does not deliver SIGKILL to the destination process. When SIGKILL is received by the UNIX process server, the UNIX process server terminates the given process.

4.5.2 At the signalled process

When a signal arrives at the signalled process via the *deliver_signal* method on the *signal_handler* object implemented by the signalled domain, *libue.so* must determine what action to take. Possible actions are:

- Ignore the signal.
- Block the signal.
- Kill the process.
- Handle the signal.

The first three actions are easy. The interesting action is handling the signal. In order to handle signals we once again use the *helper_thread* that we used for *fork*. Note that the thread that is invoking the *signal_handler* object is a new thread which we will call the *signal_thread*. In order to deliver a signal the following steps are followed:

1. The *signal_thread* stops the *main_thread*, gets the *main_thread's* registers and stores them on its stack, and then continues the *main_thread* with modified registers so that it will begin executing in a routine that will call the signal handler.
2. The *main_thread* starts executing and then calls the signal handler.
3. When the signal handler returns, the main thread gets its old registers off of the stack, stores them in a global structure, wakes up the *helper_thread*, and goes to sleep.
4. The *helper_thread* wakes up and resumes the *main_thread* with its old registers.

We also deliver signals at other times. For example, if a signal that has been blocked is suddenly enabled, then it will be delivered immediately. This is done by having routines such as *sigsetmask* check the list of pending signals before they return. If they find a pending signal that needs to be delivered then they call the signal handler directly; there is no need to save state on the stack or use the *helper_thread* because the *main_thread* can just call the signal handler itself.

4.6 Virtual memory

UNIX virtual memory calls translate easily into calls on Spring's *address_space* object, and the UNIX process

server is not involved in handling these calls. In general, the Spring virtual memory system is a super-set of UNIX virtual memory operations. An interesting exception is copy-on-write. The UNIX mmap(2) call with MAP_PRIVATE flag establishes a pseudo copy-on-write mapping, since any modifications made to the source memory object are visible to the process that establishes the private mapping as long as such writes are made before the private copy is written. Spring virtual memory on the other hand provides true copy-on-write (modifications to the source memory object or to the private copy are not visible to the other). We implemented UNIX's MAP_PRIVATE using the copy-on-write implementation of Spring despite the difference in semantics and did not encounter any applications that cared about the difference in the map semantics.

5 Implementation Status

The implementation of the UNIX process server consists of 7000 lines of C++, while *libue.so* is implemented using 14500 lines of code. The effort took approximately 1 person-year to complete. Around 60% of SunOS 4.1 system calls are implemented. The main exceptions are *ptrace*, System V IPC and stream calls, and calls such as *sigstack*, *audit*, *mknod*, and *mount*. Despite these omissions we run most SunOS binaries without modifications, including X/NeWS, emacs, vi, csh, make, and various compilers.

As we described before, we used the Spring file system, virtual memory, dynamic linking, networking, and device drivers, and we did not have to re-implement any of these basic operating system services for the UNIX system.

The base Spring system as a whole is now very stable and usable. The initial implementation of the system was for the *sun4c* architecture (SPARCstation 1, 2). The system was then ported to the *sun4m* multiprocessor architecture (SPARCstation 10 and SPARCserver 600). All system servers including the kernel are multi-threaded, and the system runs in uniprocessor and symmetric multiprocessor configurations.

The current focus of the Spring project has been on developing and implementing the basic architectures. We have spent very little time on performance evaluation and tuning the system. Since we now have a complete system, our focus is shifting towards measuring and tuning the system and evaluating how well we have achieved our performance goals. We plan to report on the performance of the system in a future report.

6 Related Work

In this section we compare our system to two other implementations of UNIX on kernelized systems: MACH 3.0 with the BSD4.3 Single Server [4] and CHORUS/MiX V.4 [5].

6.1 MACH 3.0 with the BSD4.3 Single Server

The BSD4.3 Single Server is a MACH task that contains an implementation of BSD4.3 [4]. An emulation library is loaded into the address space of UNIX processes (using virtual memory inheritance starting from /etc/init). A system call typically traps into the MACH kernel and is redirected back into the emulation library of the trapping process. The emulation library then sends a message to the BSD server which in turn executes the actual UNIX call. The BSD server shares two pages with each UNIX process, which are used to communicate some information between the server and its client.

Unlike the centralized BSD server, our UNIX process server only provides support for redirecting signals, keeps track of basic relationships among UNIX processes and provides support for pipes and local sockets. Moreover, it is not involved at all in implementing other UNIX calls, such as file system and virtual memory operations. As we mentioned before, we rely on native Spring servers for such things as the file system, virtual memory, dynamic linking, networking protocols, and naming support. However, unlike the BSD server implementation, our implementation does not currently support statically-linked executables (but see Section 7).

Our implementation does not rely on sharing memory between the UNIX process manager and UNIX processes; we believe that our implementation works better on NUMA machines. Our file system moreover provides consistent shared files across the network, and in general all the servers in our system can be located on more than one machine.

6.2 CHORUS/MIX V.4

MiX V.4 is a subsystem built on top of the CHORUS kernel [5]. MiX V.4 is composed of a set of servers that communicate through CHORUS IPC. The most important MiX V.4 server is the Process Manager (PM) through which client program UNIX calls are directed. Other servers include the File Manager (FM) and the Streams Manager (StM).

The CHORUS implementation is perhaps closer to ours in that the implementation of the various UNIX functionality is split among several CHORUS servers. An important difference, however, is that unlike our implementation all UNIX process calls in MiX V.4 have to pass through the PM on their way to their respective servers. In addition, the MiX V.4 implementation is tuned for running the various MiX servers in the supervisor address space [6] (although MiX V.4 servers are fully independent and can run in independent user spaces). We do not plan on moving our servers into supervisor space.

We share with MiX V.4 the support for network-wide shared files and the general distributed nature of the implementation. However, we only implement a subset of SunOS 4.1 whereas MiX V.4 is a complete implementation of SVR4.

7 Conclusions and Future work

We implemented a UNIX subsystem on top of a non-UNIX object-oriented operating system. As a result we were able to run a large number of existing applications on Spring. The implementation showed the flexibility of our system since we were able to achieve our goals without changing Spring. The implementation exercised the underlying system and forced us to complete some missing functionality of Spring.

We believe that a fundamental reason for our successful effort was the decision to provide an implementation at the man (2) system calls without rewriting any UNIX libraries. In doing so, we confined our effort to a (relatively) well-defined interface that *libc* and other libraries used.

There are several ways in which we can extend this work:

- **Implement the rest of the system calls.** SunOS provides a rich set of system calls. Although we only implemented a subset of the system calls, we were able to run most programs. As we gain more experience with the system we may add some of the missing functionality.
- **Provide SunOS 5.0 multi-threaded application interfaces.** The current implementation is tailored toward supporting SunOS 4.1 calls and libraries. Our work was developed in parallel with SunOS 5.0. which is based on SVR4 and provides a multi-thread application architecture [1]. Now that the 5.0 work is done we plan to port to its interfaces. We do not expect that this will be difficult since the Spring system and UNIX process server are already multi-threaded, and so is most of *libue.so*. Modifications will mainly be the extensions made in SunOS 5.0 to the signal model [1].
- **Handle statically-linked binaries.** The current implementation cannot run statically linked executables. For our purposes, we believe that it is not worth the effort to provide binary compatibility as most UNIX applications are dynamically linked. Moreover we believe that the use of dynamic linking will increase in the future. Our architecture does not preclude providing such functionality, however. Spring provides the ability to field domain traps and convert the traps into invocations on callback objects. One can provide support for statically-linked binaries by establishing a callback object on each UNIX domain (where the implementation of the callback object resides in the UNIX process itself). The implementation can then field the callbacks, decode the trap information and then use the *same libue.so* code to execute the calls.
- **Move pipes and pty implementation out of the UNIX process server.** Currently, obtaining pipes and ptys, as well as moving data through them, is done through the UNIX process server. We can get better performance by separating the functionality of setting up the connection from data movement. Such an implementation would use the UNIX process server for setting up the initial connection, but would copy the data directly between UNIX processes.
- **More efficient exec.** The current implementation of exec requires that a new domain be created whenever a process execs. Another option would be to do the exec in place; that is, replace the current domain's address space with the exec'd domains address space. This would require that a portion of code always be in each domain that can be used for this purpose.

- **Allow access to non-UNIX type objects.** Currently, UNIX domains can only access those types of objects that exist on UNIX. For example, if a stream object that were bound somewhere in the name space were opened by a UNIX domain and it wasn't a file the open would fail. We should be able to allow UNIX domains access to generic Spring objects as long as they support the *io.sequential_io* Spring interface.
- **Extend UNIX semantics.** An interesting opportunity made possible by the Spring system is to extend UNIX semantics to a distributed system. New functionality such as remote fork and exec operations, and network-wide coherent mapped memory can be added without much additional effort. Spring object invocation is location-independent, and all Spring services are distributed in nature. The ability to share memory and files across the network in a coherent fashion is already provided by Spring virtual memory and file systems.

Acknowledgments

We would like to acknowledge the efforts of Graham Hamilton and Sanjay Radia in helping to get the UNIX system running on Spring. We would like to also acknowledge the efforts of all Spring contributors whose work made this project a success.

References

[1] M. L. Powell, S. R. Kleiman, S. Barton, D. Shah, D. Stein, M. Weeks, "SunOS Multi-thread Architecture", *USENIX Winter 1991*, pp. 65-79.

[2] Sanjay Radia, "Names, Contexts, and Closure Mechanisms in Distributed Computing Environments", Ph.D. thesis, Technical report UW/ICR 90-01, Department of Computer Science, University of Waterloo, 1989.

[3] R. Pike, D. Presotto, K. Thompson, and H. Trickey, "Plan 9 from Bell Labs", *Proceedings of the Summer 1990 UKUUG Conference*, July 1990, pp. 1-9.

[4] D. Golub, R. W. Dean, A. Forin, and R. F. Rashid, "UNIX as an Application Program", *Proceedings of Summer 1990 USENIX Conference*, June 1990, pp. 87-95.

[5] N. Batlivala, et al., "Experience with SVR4 Over CHORUS", *Proceedings of USENIX Workshop on Micro-kernels and Other Kernel Architectures*, April 1992, pp. 223-241.

[6] R. W. Dean and F. Armand, "Data Movement in Kernelized Systems", *Proceedings of USENIX Workshop on Micro-kernels and Other Kernel Architectures*, April 1992, pp. 243-261.

Yousef A. Khalidi is currently a Senior Staff Engineer at Sun Microsystems Laboratories. His interests include distributed, object-oriented software, operating systems, and architecture. He has a Ph.D. in Information and Computer Science from Georgia Institute of Technology. He can be reached at Sun Microsystems Laboratories, Inc., 2550 Garcia Ave., MTV29-112, Mt. View, CA 94043, USA, or via e-mail at yak@sun.com.

Michael N. Nelson is currently a Senior Staff Engineer at Sun Microsystems Laboratories. Before joining Sun he was one of the principal developers of the Sprite Operating System at Berkeley and worked at DEC Western Research Laboratory. His interests include distributed, object-oriented software, operating systems, and architecture. He has a Ph.D. in Computer Science from UC Berkeley. He can be reached at Sun Microsystems Laboratories, Inc., 2550 Garcia Ave., MTV29-112, Mt. View, CA 94043, USA, or via e-mail at michael.nelson@sun.com.

The Nachos Instructional Operating System[1]

Wayne A. Christopher, Steven J. Procter, & Thomas E. Anderson
– University of California at Berkeley

ABSTRACT

In teaching operating systems at an undergraduate level, we believe that it is important to provide a project that is realistic enough to show how real operating systems work, yet is simple enough that the students can understand and modify it in significant ways. A number of these instructional systems have been created over the last two decades, but recent advances in hardware and software design, along with the increasing power of available computational resources, have changed the basis for many of the tradeoffs made by these systems.

We have implemented an instructional operating system, called Nachos, and designed a series of assignments to go with it. Our system includes CPU and device simulators, and it runs as a regular UNIX process. Nachos illustrates and takes advantage of modern operating systems technology, such as threads and remote procedure calls, recent hardware advances, such as RISC's and the prevalence of memory hierarchies, and modern software design techniques, such as protocol layering and object-oriented programming. Nachos has been used to teach undergraduate operating systems classes at several universities with positive results.

1 Introduction

In undergraduate computer science education, course projects provide a useful tool for teaching basic concepts and for showing how those concepts can be used to solve real-world problems. A realistic project is especially important in undergraduate operating systems courses, where many of the concepts are best taught, we believe, by example and experimentation.

This paper discusses an operating system, simulation environment, and set of assignments that we developed for the undergraduate operating systems course at the University of California at Berkeley.

Over the years, numerous projects have been developed for teaching operating systems; among the published ones are Tunis [13] and Minix [1] [36]. Many of these projects were motivated by the development of UNIX [32] in the mid-1970's. Earlier operating systems, such as MULTICS [7] and OS/360 [25] were far too complicated for an undergraduate to understand, much less modify, in a semester. Even UNIX itself is too complicated for this purpose, but UNIX showed that the core of an operating system can be written in only a few dozen pages with a few simple but powerful interfaces [23]. Indeed, the project previously used at Berkeley, the TOY Operating System, was originally developed by Ken Thompson in 1973.

The introduction of minicomputers, and later, workstations, also aided the development of instructional operating systems. Rather than having to run the operating system on the bare hardware, computing cycles became cheap enough to make it feasible to execute an operating system kernel using a simulation of real hardware. The operating system can run as a normal UNIX process, and invoke the simulator when it would otherwise access physical devices or execute user instructions. This vastly simplifies operating systems development, by reducing the compile-execute-debug cycle and by allowing the use of off-the-shelf symbolic debuggers. Because of these advantages, many commercial operating system development efforts now routinely use simulated machines [3].

However, recent advances in operating systems, hardware architecture, and software engineering have left many operating systems projects developed over the past two decades out of date. Networking and distributed applications are now commonplace. Threads are crucial for the construction of both operating systems and higher-level concurrent applications. And the cost-performance tradeoffs among memory, CPU speed and secondary storage are now quite different from those imposed by core memory, discrete logic, magnetic drums, and card readers.

For these reasons, we decided to design and implement a new teaching operating system and

[1]This work was supported in part by the National Science Foundation (CDA-8722788) and the Digital Equipment Corporation (the Systems Research Center and the External Research Program). Anderson was also supported by a National Science Foundation Young Investigator Award.

simulation environment. Our system, called Nachos, makes it possible to give assignments that require students to write significant portions of each of the major pieces of a modern operating system: thread management, file systems, multiprogramming, virtual memory, and networking. We use these assignments to illustrate principles of computer system design needed to understand the computer systems of today and of the future: concurrency and synchronization, caching and locality, the tradeoff between simplicity and performance, building reliability from unreliable components, dynamic scheduling, the power of a level of translation, layering, and distributed computing. Facility with these concepts is valuable, we believe, even for those students who do not end up working in operating system development.

In building Nachos, we were continually faced with a tradeoff between simplicity and realism in choosing what code we provided students and what we asked students to implement. A careful balance is needed between the time students spend reading code versus adding features to existing code versus learning new concepts. Starting with code that is too realistic could lead students to lose sight of key ideas in a forest of details.

Our approach was to build the simplest implementation we could think of for each sub-system of Nachos; this provides students a working example, albeit overly simplistic, of the operation of each component of an operating system. The assignments ask the students to add functionality to this bare-bones system and to improve its performance on micro-benchmarks that we provide. As a result of our emphasis on simplicity, the Nachos operating system is about 2500 lines of code, about half of which are devoted to interface descriptions and comments.[2] It is thus practical for students to read, understand, and modify Nachos during a single semester course. By contrast, the UNIX BSD 4.3 file system *by itself*, even excluding the device drivers, is roughly 5000 lines of code [20]. Since we spend only about two to three weeks of the semester on file systems, this makes UNIX impractical as a basis for an undergraduate operating systems course project.

The first version of Nachos was completed in January 1992 and used for one term as the project for the undergraduate operating systems course at Berkeley. We then revised both the code and the assignments, releasing the second version of Nachos for public distribution in August 1992. This version is currently in use at several universities including Stanford, Harvard, Carnegie-Mellon, Colorado State, University of Washington, and of course, Berkeley; this paper focuses on describing this version. We are also continuing to work to further improve Nachos. Nachos currently runs on both DEC MIPS and SUN SPARC workstations; we believe that it would be straightforward to port Nachos to other platforms.

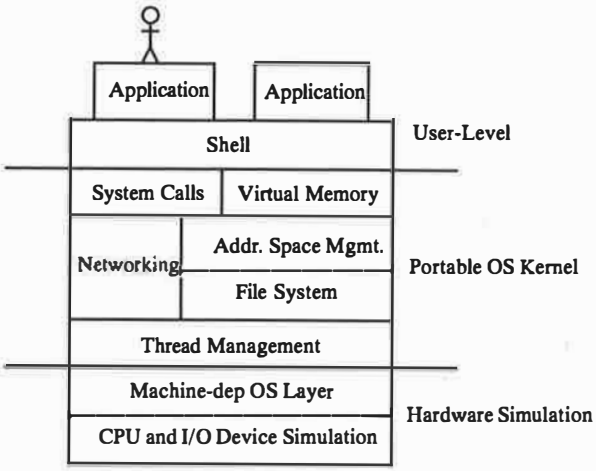


Figure 1: Nachos Software Structure

The rest of this paper describes Nachos in more detail. Section 2 provides an overview of Nachos; Section 3 describes the Nachos assignments. Sections 4 and 5 summarize our experiences.

2 Nachos Overview

Figure 1 outlines the internal structure of the Nachos instructional software. In Nachos, as in many of its predecessor systems, applications, the operating system kernel, and the hardware simulator run together in a normal UNIX process.[3]

In this UNIX process, at the lowest level, Nachos simulates the behavior of a standard uniprocessor workstation, including CPU instruction execution, address translation, interrupts, and several physical I/O devices, such as a disk, a network controller, and a console. The Nachos operating system kernel runs on top of the hardware simulation, providing many of the standard features of a modern operating system kernel, including threads, a file system, and virtual memory support. User-level applications, such as the shell, run on top of this kernel via a traditional system call interface. For efficiency, the hardware simulation and the operating system kernel run in native mode, at full speed on the underlying physical hardware; we simulate instruction execution only for user-level application code, to allow us to catch user-level page faults and other exceptions.

[2]The hardware simulator takes up another 2500 lines, but students do not need to understand the details of its operation.

[3]By contrast, Minix runs directly on personal computer hardware, avoiding the need for simulation. While this approach is more realistic, it has the disadvantage of making debugging more difficult.

Nachos has several significant differences with earlier systems:

- We can run normal C programs as user programs on our operating system, because we simulate a standard, well-documented, instruction set (MIPS R2/3000 integer instructions [15]). In the past, operating systems projects typically simulated their own *ad hoc* instruction set, requiring user programs to be written in a special purpose assembly language. Because the R2/3000 is a RISC, our instruction set simulation code is only about 10 pages long.
- We accurately simulate the behavior of a network of workstations, each running Nachos. We connect Nachos "machines", each running as a UNIX process, together via UNIX sockets, simulating a local area network. A thread on one "machine" can then send a packet to a thread running on a different "machine"; of course, both are simulated on the same physical hardware.
- Our simulation is deterministic. Debugging non-repeatable execution sequences is a fact of life for professional operating system engineers, but it did not seem advisable for us to make this experience our students' first introduction to operating systems. Instead of using UNIX signals to simulate asynchronous devices such as the disk and the timer, Nachos maintains a simulated time that is incremented whenever a user program executes an instruction and whenever a call is made to certain low-level operating system routines. Interrupt handlers are then invoked when the simulated time reaches the appropriate point.[4]
- Our simulation is randomizable to add unpredictable, but repeatable, behavior. For instance, the network simulation randomly chooses which packets to drop; provided the initial seed to the random number generator is the same, however, the behavior of the system is repeatable.
- We hide our hardware simulation routines from the rest of Nachos via a machine-dependent interface layer [31]. For example, we define an abstract disk that accepts requests to read and write disk sectors and provides an interrupt handler to be called on request completion. The details of our disk simulator are hidden behind this abstraction, in much the same way that disk device specific details are isolated in a normal operating system. One advantage to using a machine-dependent interface layer is to help students understand that they are building a real operating system: the Nachos kernel could be ported to a physical machine simply by replacing the hardware simulation with real hardware and some machine-dependent driver routines. Another advantage is to make clear to students what portions of Nachos can be modified (the kernel and the applications) versus what portions are off limits (the hardware simulation – at least until they take a computer architecture course). We did not make this distinction clear in our first version of Nachos, to our later regret.
- Nachos is implemented in a subset of C++ [34]. Object-oriented programming is becoming more popular, and we found that it was a natural idiom for stressing the importance of modularity and clean interfaces in building operating systems. To simplify matters, we omitted certain aspects of the C++ language: derived classes, operator and function overloading, C++ streams, and generics. We also kept inlines to a minimum. Although our students did not know C++ before taking our course, we found that they learned our subset of the language very easily.
- The Nachos assignments take a quantitative approach to operating system design. Frequently, the choice of how to implement some piece of operating system functionality comes down to a tradeoff between simplicity and performance. We believe that teaching students how to make informed decisions about tradeoffs is one of the key roles of an undergraduate operating systems course. The Nachos hardware simulation reflects current hardware performance characteristics; we exploit this by having students measure and explain the performance of their implementations on some simple benchmarks that we provide.

[4]The one aspect of the simulation we did not make reproducible was the precise timing of network communications. Since this came at the end of the semester, it did not seem to cause problems. We are working on providing precise network timing for the next release of Nachos.

3 The Assignments

Nachos contains five major components, each the focus of one assignment given during the semester: thread management and synchronization, the file system, user-level multiprogramming support, the virtual memory system, and networking. Each assignment is designed to build upon previous ones; for instance, every part of Nachos uses thread primitives for managing concurrency. This reflects part of the charm of developing operating systems: you get to "use what you build."

In this section, we discuss each of the five assignments in turn, describing the hardware simulation facilities and the operating system structures we

provide, along with what we ask the students to implement. The assignments are intended to be of roughly equal size, each taking 3 weeks of a 15 week semester course. The file system assignment appears from two semesters experience to be the hardest of the five; the multiprogramming assignment seems to give students the least difficulty. We spend on average 30-45 minutes in section each week discussing the assignments. Students work in pairs, and we conduct 15 minute graded design reviews after every assignment with each team. We found that the design reviews were very helpful at encouraging students to design before implementing.

Thread Management

The first assignment introduces the concepts of threads and concurrency. We provide students with a basic working thread system and an implementation of semaphores; the assignment is to implement Mesa-style locks and condition variables [18] using semaphores, and then to implement solutions to a number of concurrency problems using these synchronization primitives [5]. For instance, we ask students to program a simple producer-consumer interaction through a bounded buffer, using condition variables to denote the "buffer empty" and "buffer full" states.

In much the same way as pointers for beginning programmers, understanding concurrency requires a conceptual leap on the part of students. Contrary to Dijkstra [8], we believe that the best way to teach concurrency is with a "hands-on" approach. Nachos helps in two ways. First, thread management in Nachos is *explicit*: students can trace, literally statement by statement, what happens during a context switch from one thread to another, both from the perspective of an outside observer and from that of the threads involved. We believe this experience is crucial to de-mystifying concurrency. Precisely because C and C++ allow nothing to be swept under the covers, concurrency may be easier to understand (although harder to use) in these programming languages than in those explicitly designed for concurrency, such as Ada [26], Modula-3 [27], and Concurrent Euclid [13].

Second, a working thread system, as in Nachos, allows students to practice writing concurrent programs and to test out those programs. Even experienced programmers find it difficult to think concurrently; a widely used OS textbook had an error in one of its concurrent algorithms that went undetected for several years. When we first used Nachos, we omitted many of the practice problems we now include in the assignment, thinking that students would see enough concurrency in the rest of the project. In retrospect, the result was that many students were still making concurrency errors even in the final phase of the project.

Our thread system is based on FastThreads [2]. Our primary goal was simplicity, to reduce the effort required for students to trace the behavior of the thread system. Our implementation takes a total of about 10 pages of C++ and a page of MIPS assembly code. For simplicity, thread scheduling is normally non-preemptive, but to emphasize the importance of critical sections and synchronization, we have a command-line option that causes threads to be time-sliced at "random", but repeatable, points in the program. Concurrent programs are correct only if they work when "a context switch can happen at any time."

File Systems

Real file systems can be very complex artifacts. The UNIX file system, for example, has at least three levels of indirection – the per-process file descriptor table, the system-wide open file table, and the in-core inode table – before one even gets to disk blocks [24]. As a result, in order to build a file system that is simple enough for students to read and understand in a couple of weeks, we were forced to make some hard choices as to where to sacrifice realism.

We provide a basic working file system stripped of as much functionality as possible. While the file system has an interface similar to that of UNIX [32] (cast in terms of C++ objects), it also has many significant limitations with respect to commercial file systems: there is no synchronization (only one thread can access the file system at a time), files have a very small maximum size, files have a fixed size once created, there is no caching or buffering of file data, the file name space is completely flat (there is no hierarchical directory structure), and there is no attempt at providing robustness across machine and disk crashes. As a result, our basic file system takes only about 15 pages of code.

The assignment is first, to correct some of these limitations, and second, to improve the performance of the resulting file system. We list a few possible optimizations, such as caching and disk scheduling, but it is up to the students to decide which are the most cost-effective for our benchmark (the sequential write and then read of a large file).

At the hardware level, we provide a disk simulator, which accepts "read sector" and "write sector" requests and signals the completion of an operation via an interrupt. The disk data is stored in a UNIX file; read and write sector operations are performed using normal UNIX file reads and writes. After the UNIX file is updated, we calculate how long the simulated disk operation should have taken (from the track and sector of the request), and set an interrupt to occur that far in the future. Read and write sector requests (emulating hardware) return immediately; higher level software is responsible for waiting until the interrupt occurs.

We made several mistakes along the way in developing the Nachos file system. In our first attempt, the file system was much more realistic than the current one, but it also took more than four times as much code. We were forced to re-write it to cut it down to something that students could quickly read and understand. When we handed out this simpler file system, we did not provide enough code for it to be completely working, leaving out file read and file write to be written by students as part of the assignment. Although these are fairly straightforward to implement, the fact that our code did not work meant that students had difficulty understanding how each of the pieces of the file system fit together.

We also initially gave students the option of which limitations to fix; from our experience, we found that students learned the most from fixing the first four listed above. In particular, the students who chose to implement a hierarchical directory structure found that although it was conceptually simple, the implementation required a relatively large amount of code.

Finally, many modern file systems include some form of write-ahead logging [11] [16] or log-structure [33], simplifying crash recovery. The assignment now completely ignores this issue, but we are currently looking at ways to do crash recovery by adding simple write-ahead logging code to the baseline Nachos file system. As it stands, the choice of whether or not to address crash recovery is simply a tradeoff. In the limited amount of time available, we ask students to focus on how basic file systems work, how the file abstraction allows disk data layout to be radically changed without changing the file system interface, and and how caching can be used to improve I/O performance.

Multiprogramming

In the third assignment, we provide code to create a user address space, load a Nachos file containing an executable image into user memory, and then to run the program. Our initial code is restricted to running only a single user program at a time. Students expand on this base to support multiprogramming. Students implement a variety of system calls (such as UNIX fork and exec) as well as a user-level shell. We also ask them to optimize the multiprogramming performance of their system on a mixed workload of I/O- and CPU-bound jobs.

While we supply relatively little Nachos code as part of this assignment, the hardware simulation does require a fair amount of code. We simulate the entire MIPS R2/3000 integer instruction set and a simple single-level page table translation scheme. (For this assignment, a program's entire virtual address space must be mapped into physical memory; true virtual memory is left for assignment four.) In addition, we provide students an abstraction that hides most of the details of the MIPS object code format.

This assignment requires few conceptual leaps, but it does tie together the work of the previous two assignments, resulting in a usable, albeit limited, operating system. Because our simulator can run C programs, our students found it easy to write the shell and other utility programs (such as UNIX "cat") to exercise their system. (One overly ambitious student attempted to port emacs.) The assignment illustrates that there is little difference between writing user code and writing operating system kernel code, except that user code runs in its own address space, isolating the kernel from user errors.

One important topic we chose to leave out (again, as a tradeoff against time constraints) is the trend toward a small-kernel operating system structure, where pieces of the operating system are split off into user-level servers [37]. Because of its modular design, it would be straightforward to move Nachos towards a small-kernel structure, except that (i) we have no symbolic debugging support for user programs and (ii) we would need a stub compiler to make it easy to make procedure calls across address spaces [4].

Virtual Memory

Assignment four asks students to replace their simple memory management code from the previous assignment with a true virtual memory system, that is, one that presents to each user program the abstraction of an (almost) unlimited virtual memory size by using main memory as a cache for the disk. We provide no new hardware or operating system components for this assignment.

The assignment has three parts. First, students implement the mechanism for page fault handling – their code must catch the page fault, find the needed page on disk, find a page frame in memory to hold the needed page (writing the old contents of the page frame to disk if it is dirty), read the new page from disk into memory, adjust the page table entry, and then resume the execution of the program. This mechanism can take advantage of what the students have built in previous assignments: the backing store for an address space can be simply represented as a Nachos file, and synchronization is needed when multiple page faults occur concurrently.

The second part of the assignment is to devise a policy for managing the memory as a cache – for deciding which page to toss out when a new page frame is needed, in what circumstances (if any) to do read-ahead, when to write unused dirty pages back to disk, and how many pages to bring in before initially starting to run a program [20] [21].

These policy questions can have a large impact on overall system performance, in part because of the large and increasing gap between CPU speed and disk latency – this gap has widened by two orders of

magnitude in only the last decade. Unfortunately, the simplest policies often have unacceptable performance. To encourage students to implement realistic policies, the third part of the assignment is to measure the performance of the paging system on a benchmark we provide – a matrix multiply program where the matrices do not fit in memory. This workload is clearly not representative of real-life paging behavior, but it is simple enough that students can understand the impact of policy changes on the application. Further, the application illustrates some of the problems with caching – small changes in the implementation of matrix multiply can have a large impact on performance [17].

Networking

Although distributed systems have become increasingly important commercially, most instructional operating systems do not have a networking component. To address this, the capstone of the project is to write a significant and interesting distributed application.

At the hardware level, each UNIX process running Nachos represents a uniprocessor workstation. We simulate the behavior of a network of workstations by running multiple copies of Nachos, each in its own UNIX process, and by using UNIX sockets to pass network packets from one Nachos "machine" to another. The Nachos operating system can communicate with other systems by sending packets into the simulated network; the transmission is actually accomplished by socket send and receive. The Nachos network provides unreliable transmission of limited-size packets from machine to machine. The likelihood that any packet will be dropped can be set as a command-line option, as can the seed used to determine which packets are "randomly" chosen to be dropped. Packets are dropped but never corrupted, so that checksums are not required.

To show students how to use the network and, at the same time, to illustrate the benefits of layering, we built a simple post office protocol on top of the network. The post office layer provides a set of "mailboxes" that serve to route incoming packets to the appropriate waiting thread. Messages sent through the post office also contain a return address to be used for acknowledgements.

The assignment is first to provide reliable transmission of arbitrary-sized packets, and then to build a distributed application on top of that service. Supporting arbitrary-sized packets is straightforward – one need merely to split any large packet into fixed-size pieces, add fragment serial numbers, and send them one by one. Reliability is more interesting, requiring a careful analysis and design to be implemented correctly. To reduce the time to do the assignment, we do not ask students to implement congestion control or window management, although of course these are important issues in protocol design [12].

The choice of how to complete the project is left up to the students' creativity. We do make a few suggestions: multi-user UNIX talk, a distributed file system with caching [28], a process migration facility [9], distributed virtual memory [22], a gateway protocol that is robust to machine crashes. Perhaps the most interesting application a student built was a distributed version of the "battleship" game, with each player on a different machine. This illustrated the role of distributed state, since each machine kept only its local view of the gameboard; it also exposed several performance problems in our hardware simulation which we have since fixed.

Perhaps the biggest limitation of our current implementation is that we do not model network performance correctly, because we do not keep the timers on each of the Nachos machines synchronized with one another. We are currently working on addressing this problem, using distributed simulation techniques for efficiency [6] [14]. With this, we will be able to benchmark the performance of the students' network protocols; this will also enable students to implement parallel algorithms for message-passing multiprocessors as the final part of the project.

4 Lessons Learned

Designing and implementing Nachos taught us a lot about how instructional software should be put together, and provided insights on how students learn about complex systems. In this section, we discuss some of the lessons that we learned.

In devising the assignments, we had to decide which pieces of the Nachos code to provide students and which pieces to leave for students to write themselves. At one extreme, we could have provided students only the hardware simulation routines, leaving a *tabula rasa* for students to build an entire operating system from scratch. This seemed impractical, given the scope of what we wanted students to achieve during the semester.

Since our goal was to maximize learning for the amount of student effort expended, we at first provided students with the mundane and the technically difficult parts of the operating system, such as generic list and bitmap management routines on the one hand, and low level thread context switch code on the other. We did this by writing the entire operating system from scratch, and then ripping out the parts that we thought students should write for themselves.

We found, however, that code (if simple enough) can be very useful at illustrating how some piece of the operating system should behave. The key is that the code has to be able to run standalone, without further effort on the part of students. Our

thread system, although limited, could show exactly what happens when one thread relinquishes a processor to another thread. By contrast, when we provided students with less than a working file system, students had difficulty understanding how the pieces of the file system fit together. Similarly, we initially left to students the definition of the system call interface, including how parameters were to be passed from user code to the kernel. A simple example would have avoided the resulting confusion.

Of course, reading code by itself can be a boring and pointless exercise; we addressed this by keeping our code as simple as possible, and by asking students to modify it in fairly fundamental ways. The result is that the assignments focus on the more interesting aspects of operating systems, where tradeoffs exist so that there is no single right answer.

Another lesson that we learned from experience was the need to add a quantitative aspect to the assignments. We explicitly encouraged students to implement simple solutions to the assignments, to avoid sprawling complexity. But because we initially had no standard benchmarks for measuring the performance of student implementations, students tended to devise overly simplistic solutions, where only a bit more effort was needed to be realistic. We hope that the performance tests that we have since added will encourage students to identify when complexity is justified by its benefits. In the future, we also intend to experiment with a different approach towards this same end – to ask students to explain what performance they would *expect* from their implementation, along with the likely effect of different performance optimizations, on a simple benchmark. The idea would be to encourage students to reason about the performance of their system, instead of simply making changes and measuring the result.

Finally, we were not able to find a textbook to adequately explain many of the concepts used in Nachos, particularly in the areas of concurrency and networking. For instance, the operating system textbook we ended up using only lightly touches on locks and condition variables; instead, it devotes most of a chapter to describing how to build critical sections using only memory read and memory write operations as primitives. Yet *every* operating system that we know of implements critical sections using interrupt disable and/or memory read-modify-write instructions.

To address this, we supplemented the textbook with a few relevant papers, namely: Birrell [5], Ritchie and Thompson [32], McKusick, et al. [24], Gray [10], Levy and Lipman [21], Hedrick [12], and Lampson [19]. We found that many of our students could understand and use the key ideas from these papers, particularly when we gave them a roadmap to each paper's terminology. An important side goal was to de-mystify reading research papers – one way for students to continue their education after graduation to keep up with the rapid pace of technological change in our industry.

5 Conclusions

We have written an instructional operating system, called Nachos. It is designed to reflect recent advances in hardware and software technology, to illustrate modern operating system concepts, and, more broadly, to help teach students how to design complex computer systems. Nachos has been used in undergraduate operating systems courses at several universities, and the results were positive. We plan to use Nachos in future semesters, and we have made it publicly available in the hope that others will also find it useful.

6 Acknowledgements

We would like to thank the Spring 1992 CS 162 class at Berkeley for serving as guinea pigs while Nachos was under development. Brian Bershad, Garth Gibson, Ed Lazowska, John Ousterhout, and Dave Patterson gave us very helpful advice during the design of Nachos. John Ousterhout also wrote the MIPS simulator included in Nachos. Mendel Rosenblum ported Nachos to the SPARC; Miguel Valdez and Yan Or are continuing work on improving Nachos. We credit Lance Berc with the acronym for Nachos: Not Another Completely Heuristic Operating System.

7 Availability

A copy of Nachos can be obtained by anonymous ftp from ftp.cs.berkeley.edu, file ''/ucb/nachos/nachos-2.1.tar.Z''. Questions about Nachos can be directed via e-mail to Anderson (tea@cs.berkeley.edu) or posted to the ''alt.os.nachos'' newsgroup.

8 References

[1] Aguirre, G., Errecalde, M., Guerrero, R., Kavka, C., Leguizamon, G., Printista, M., and Gallard, R. Experiencing MINIX as a Didactical Aid for Operating Systems Courses. *Operating Systems Review*, 25:32-39, July 1991.

[2] Anderson, T., Lazowska, E., and Levy, H. The Performance Implications of Thread Management Alternatives for Shared Memory Multiprocessors. *IEEE Transactions on Computers*, 38(12):1631-1644, December 1989.

[3] Bedichek, R. Some Efficient Architecture Simulation Techniques. In *Proceedings of the 1990 USENIX Winter Conference*, pp. 53-63, January 1990.

[4] Birrell, A. and Nelson, B. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39-59, February 1984.

[5] Birrell, A. An Introduction to Programming with Threads. Technical Report #35, Digital Equipment Corporation's Systems Research Center, Palo Alto, California, January 1989.

[6] Chandy, K. and Misra, J. Asynchronous Distributed Simulation via a Sequence of Parallel Computations. *Communications of the ACM*, 24(4):198-206, April 1981.

[7] Daley, R. and Dennis, J. Virtual Memory, Processes and Sharing in MULTICS. *Communications of the ACM*, 11(5):306-312, May 1968.

[8] Dijkstra, E. On the Cruelty of Really Teaching Computer Science. *Communications of the ACM*, 32(12):1398-1404, December 1989.

[9] Douglis, F. and Ousterhout, J. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software–Practice and Experience*, 21(7), July 1991.

[10] Gray, J. The Transaction Concept: Virtues and Limitations. In *Proceedings of the 7th International Conference on Very Large Data Bases*, pp. 144-154, September 1981.

[11] Hagmann, R. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pp. 155-162, November 1987.

[12] Hedrick, C. Introduction to the Internet Protocols. Technical report, Rutgers Computer Science Facilities Group, July 1987.

[13] Holt, R. *Concurrent Euclid, the UNIX System, and TUNIS*. Addison-Wesley, 1983.

[14] Jefferson, D., Beckman, B., Wieland, F., Blume, L., DiLoreto, M., Hontabas, P., Laroche, P., Studevant, K., Tupman, J., Warren, V., Wedel, J., Younger, H., and Bellenot, S. Distributed Simulation and the Time Warp Operating System. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pp. 77-93, November 1987.

[15] Kane, G. *MIPS R2000 RISC Architecture*. Prentice Hall, 1987.

[16] Kazar, M., Leverett, B., Anderson, O., Apostolides, V., Bottos, B., Chutani, S., Everhart, C., Mason, W., Tu, S.-T., and Zayas, E. DEcorum File System Architectural Overview. In *Proceedings of the 1990 USENIX Summer Conference*, pp. 151-164, June 1990.

[17] Lam, M., Rothberg, E., and Wolf, M. The Cache Performance and Optimizations of Blocked Algorithms. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 63-74, April 1991.

[18] Lampson, B. and Redell, D. Experiences with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):104-117, February 1980.

[19] Lampson, B. W. Hints for Computer System Design. *IEEE Software*, 1(1):11-28, Jan 1984.

[20] Leffler, S., McKusick, K., Karels, M., and Quarterman, J. *Design and Implementation of the 4.3 BSD Unix Operating System*. Addison-Wesley, 1989.

[21] Levy, H. and Lipman, P. Virtual Memory Management in the VAX/VMS Operating System. *IEEE Computer*, pp. 35-41, March 1982.

[22] Li, K. and Hudak, P. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321-359, November 1989.

[23] Lions, J. A Commentary on the UNIX Operating System, June 1977. Department of Computer Science, University of New South Wales.

[24] McKusick, M., Joy, W., Leffler, S., and Fabry, R. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181-197, August 1984.

[25] Mealy, G., Witt, B., and Clark, W. The Functional Structure of OS/360. *IBM Systems Journal*, 5(1):3-51, January 1966.

[26] Mundie, D. and Fisher, D. Parallel Processing in Ada. *IEEE Computer*, 19(8):20-25, August 1986.

[27] Nelson, G., editor. *Systems Programming with Modula-3*. Prentice Hall, 1991.

[28] Nelson, M., Welch, B., and Ousterhout, J. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1):134-154, February 1988.

[29] Patterson, D. and Hennessy, J. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, San Mateo, CA, 1990.

[30] Patterson, D. Has CS Changed in 20 Years? *Computing Research News*, 4(2):2-3, March 1992.

[31] Rashid, R., Tevanian, A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W., and Chew, J. Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures. *IEEE Transactions on Computers*, 37(8):896-908, August 1988.

[32] Ritchie, D. and Thompson, K. The Unix Time-Sharing System. *Communications of the ACM*, 17(7):365-375, July 1974.

[33] Rosenblum, M. and Ousterhout, J. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26-52, February 1992.

[34] Stroustrup, B. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1986.

[35] Tanenbaum, A. *Operating Systems: Design and Implementation*. Prentice-Hall, 1987.

[36] Tanenbaum, A. A UNIX Clone with Source Code for Operating Systems Courses. *Operating Systems Review*, 21(1):20-29, January 1987.

[37] Wulf, W., Cohen, E., Corwin, W., Jones, A., Levin, R., Pierson, C., and Pollack, F. HYDRA: The Kernel of a Multiprocessor Operating System. *Communications of the ACM*, 17(6):337-344, June 1974.

Author Information

Wayne Christopher is a graduate student in the Computer Science Division at the University of California at Berkeley. He received his B.A. in mathematics and philosophy in 1986 and his M.S. in electrical engineering in 1989, both from Berkeley. With any luck he will receive his Ph.D. in 1993. His research interests include realistic animation, virtual reality, multimedia, networks, and operating systems. Reach him electronically at faustus@cs.berkeley.edu.

Steven Procter is a senior software engineer at Real Time Solutions in Berkeley, California. He received his B.A. in mathematics in 1988 and his M.S. in computer science in 1992, both from the University of California at Berkeley. His interests include operating systems, two dimensional computer graphics and multimedia. His e-mail address is rts2!procter@uunet.uu.net.

Thomas Anderson is an Assistant Professor in the Computer Science Division at the University of California at Berkeley. He received his A.B. in philosophy from Harvard University in 1983 and his M.S. and Ph.D. in computer science from the University of Washington in 1989 and 1991, respectively. He won an NSF Young Investigator Award in 1992, and he co-authored award papers at the 1989 SIGMETRICS Conference, the 1989 and 1991 Symposia on Operating Systems Principles, and the 1992 ASPLOS Conference. His interests include operating systems, computer architecture, multiprocessors, high speed networks, massive storage systems, and computer science education. His e-mail address is tea@cs.berkeley.edu.

The Design and Implementation of a Mobile Internetworking Architecture

John Ioannidis & Gerald Q. Maguire Jr. – Columbia University

ABSTRACT

We present the design, implementation, and evaluation of **Mobile*IP**, a set of IP-based protocols and mechanisms to support host mobility throughout the Internet. The design requires changes only in the mobile hosts and their special routers; leaves transport and higher protocols unaffected, and requires no changes in the device drivers for individual interfaces. No modifications whatsoever are needed in non-mobile hosts and routers, the system scales well, and has no single points of failure. We have implemented Mobile*IP under Mach 2.6, and the code is readily portable to any version of Unix that uses Berkeley networking code.

1 Introduction

Motivation

The continuing drop in prices and increase in functionality of personal, portable computers, the increasing availability of wireless networking options as well as wide-area research and commercial networking offerings, and an increased desire of users to carry these systems and connections with them while they travel, suggests a marketplace and a user base ripe for introducing transparent network mobility to existing networking architectures. The proliferation of terms such as *Nomadic Computing*, *Personal Communications Networks*, [2, 4, 13, 20, 21, 23], as well as the rapid expansion of more established technologies such as the Cellular Phone System [11], pagers, the new cordless phones [22], etc., suggest that the mobile/wireless industry is moving fast. However, there is no clear sense of what services should be supported or what infrastructure is required.

We expect that in a few years, wireless/mobile support will be as widespread in educational, research, and business environments as Ethernet or other LAN connections are today. Rather than trying to define and predict specific applications, we looked at the general problem of mobile data communications and designed a solution that works across a wide variety of technologies and applications, and interoperates with the Internet. Our design was first described in [8] and fully specified in [9], and a reference implementation is freely available.

This paper documents our implementation, providing a reasonable amount of detail of the software structure, and evaluates its performance. It is also intended to serve as an guide, in conjunction with [9], to other people wishing to implement Mobile*IP on their platforms.

In the remainder of this section, we outline our design goals, and give a summary overview of how our system works. The next Section discusses the rationale behind the particular addressing and routing mechanisms that we chose. Sections 3 and 4 describe the software design, implementation details, and performance, in the Mobile Support Routers and Mobile Hosts, respectively. Section 5 describes the additional software components needed for Popups (that is, Mobile Hosts migrating outside their home networks), and Section 6 completes the paper.

Design Goals

The concept of routing for mobile hosts is not a new one [7, 10, 12, 24]; these previous designs, however, are impractical in today's Internet, with its vast number of applications, hosts, and networking infrastructure. We have developed a new approach, optimized for localized mobility, driven by the following design goals:

- Work within the TCP/IP protocol suite [17] [18].
- Provide Internet-wide mobility.
- A mobile host always keeps its IP address, called its "Home Address".
- Optimize local-area mobility without sacrificing performance or functionality of the general case.
- Transport-layer and higher protocols should be left untouched.
- No applications should change in order to run on or be used from mobile hosts.
- The infrastructure, that is, non-mobile hosts, routers, routing protocols, etc. should be left untouched.
- Mobility should be handled at the network layer.
- Minimize points of failure.
- Be responsive and scale well.

The rationale behind those choices is presented in the aforementioned papers. In summary, it is crucially important for mobile hosts to maintain their IP address as they move, since their IP address is also used as the Endpoint Identifier (EID) [15] for connections, and trying to notify transport protocols and applications that the host address has changed was deemed unrealistic. In addition, we did not expect our approach to be adopted if it required ordinary hosts and routers to be modified in order to talk to mobile machines. Lastly, we wanted to achieve mobility with the least amount of impact to the rest of the network. These goals were met by decoupling routing to/from the mobile hosts from routing to/from the rest of the network; by requiring the Mobile Hosts (MHs) to have IP addresses from a reserved subnet or subnets; and having a special class of routers, called the Mobile Support Routers (MSRs), route packets between the MHs and the rest of the network.

Overview of Operations

Figure 1 shows a small portion of a campus network that supports mobile hosts. Each MSR defines one or more cells, one for each network interface. The cells may be wireless (RF or IR), wired (Ethernet, token-ring, FDDI) or just groups of point-to-point links where mobiles can attach. In the example shown in Figure 1, there is only one cell per MSR, and all mobiles have RF interfaces. Three MSRs, one MH, and one non-mobile host are shown, as well as the routers (R and Rw) linking the three segments together and to the outside world.

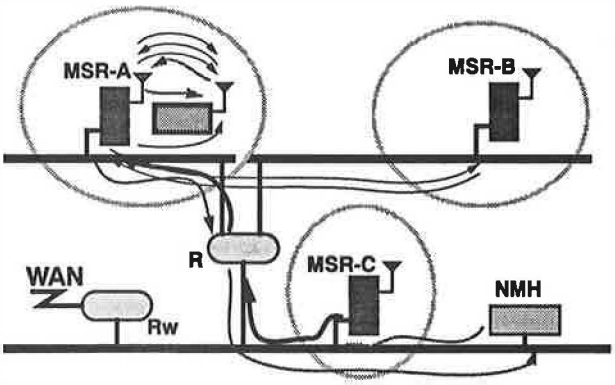


Figure 1: Sample Campus.

In each cell, the MSR broadcasts a periodic Beacon. MHs entering the cell receive the Beacon, send a Greeting to the MSR, which answers with an Greeting Acknowledgement (as shown by the three thin solid arrows between the mobile and MSR-A. If the mobile was previously in MSR-B's cell, it will also send a Forwarding Pointer (FWDPTR) notifying it of its migration, which should be acknowledged with a Forwarding Acknowledgement (FWDACK). Once in MSR-A's cell, the MH sets it as its default router to the world, and MSR-A marks the MH as "local". IP datagrams from an MH to another MH in the same cell are processed locally. IP datagrams from an MH to a non-mobile host anywhere else on the Internet are routed in the usual manner. IP datagrams from a non-mobile host are sent to the "nearest" MSR; if the target MH is served by that MSR, the datagram is simply forwarded to the MH. Otherwise, if the MSR knows which MSR is currently handling the MH, it encapsulates the datagram in another datagram (shown by the change of gray density in the arrow going from NMH through MSR-C, and on to MSR-A via the router R, and sends it to that other MSR, which decapsulates it and delivers it to the MH. If the original MSR (in the example, MSR-C) does not know where the MH is, it queries the other MSRs (it can also query a name-location server), and the one handling the MH will respond; the response is cached as long as there is traffic to the target MH, and future datagrams do not cause a new query. Finally if an MH sends a datagram to an MH in a different cell, the first MH's MSR will receive it and tunnel the datagram to the MSR handling the other MH.

Mobiles that wish to connect from outside their home network and still appear to be hosts belonging to that network, acquire a temporary address, called the Nonce Address, in the foreign subnet, then use it to first handshake with one of its home MSRs, and then tunnel packets to and from an MSR from their home network.

2 Network Considerations

The main problem of trying to add mobility to IP, is that the IP address of a host is, at once, an *Endpoint Identifier (EID)*, that is, a (unique) name used to identify the connections to and from the host, and also an *address*, that is, an indication of *where* the host is, and thus *how* to reach it (route packets to it). The address has structure (`<network, subnet, [subnet...], host>`) to make routing scale. This structure, however, implies that the IP address of a host is determined by the subnet in which it is connected; ordinarily, if it were to move to another subnet, it would have to change its IP address. Since such changes were deemed unacceptable, we came up with a design that avoids this problem. In this section, we discuss the design alternatives and the reason behind our choice.

Routing and Addressing Architecture

In order to allow hosts to move within an administrative domain, (e.g., a business network or a university campus) without changing their IP address, some aspect of the nature of their IP address would have to be changed. The options were:

- Discard the 'unique-identifier' aspect. That is, when a host migrates, it is assigned a new IP address. Transport-level and higher protocols, including applications, would need to deal with such changes, and therefore this is not an acceptable option.
- Flatten the structure, at least for the mobile hosts, which would imply distributing *per-host* routes in all routers in the campus. As we shall see in Section 2, this is prohibitively expensive for large networks
- Extend the notion of 'subnet' to convey not just a connected network, but also a Virtual subnet which consists of a set of partitioned physical networks (which we shall call *cells*) linked with tunnels. The mobile hosts belong to this virtual network.

In the third option, all Mobile Hosts have addresses in the same logical IP subnet, the Mobile Subnet. This is the so-called 'embedded network' approach [1]. Routes to the Mobile Subnet are via the Mobile Support Routers; thus, to route to a mobile, it is necessary to first route to the nearest MSR, which will tunnel the datagram to the mobile's MSR, which will subsequently deliver the datagram to the target mobile. This way, only the MSRs have to know where each mobile is (as opposed to *every* router), thus reducing the amount of routing updates necessary. Tunneling is necessary because the datagrams may have to traverse multiple routers, and eventually an MSR, in order to reach their destination, and these intermediate routers do not know how to route to the particular mobile; all they know is how to route to the mobile subnet.

MSRs advertise routes to the Mobile Subnet using ordinary Internal Routing Protocols, such as RIP[5], Hello[14], IGRP[6], etc. In selecting the routing protocol and the redistribution parameters of the route(s) to the Mobile Subnet, care must be taken to ensure that the nearest active MSR is always used as an entry router to the Mobile Subnet; poor configuration may result in all the campus routers routing traffic to exactly one MSR, usually as a result of having the internal routers trust each other more than the MSRs as far as routing updates are concerned.

Each MSR 'supports' one or more cells. Since any mobile may be in any cell, all cells have the same subnet number. The result of this architecture is that the mobile subnet is really comprised of many unconnected physical network segments, the cells. In order to make this partitioned network appear as a single subnet, the MSRs exchange information about which mobiles are where, and *tunnel* datagrams between them when they are required to route a datagram destined for a mobile in another MSR's cell.

Two protocols were defined for the purposes of supporting Internet Mobility: The IP-inside-IP Encapsulation protocol (IPIP), and the Mobile Internetworking Control Protocol (MICP). Their numbers, assigned by the Internet Assigned Numbers Authority [19] are as follows:

```
#define IPPROTO_IPIP    94
#define IPPROTO_MICP    95
```

Encapsulation is a common technique for delivering data to a remote endpoint through routers that do not know how to route to that endpoint. In the case of Mobile*IP, encapsulation is used between MSRs to deliver IP datagrams whose destination address is a mobile served by the target MSR. The alternative to encapsulation is to use source routing to first route to the MSR handling the target MH, and from there deliver the packet to its destination. Source routing has, however, several problems; it is an IP option, and as such, it needs extra processing in every router the packet goes through; it interferes with a potentially already present SSRR, LSRR, or RR option; the transmitted IP packet may not have enough space left in the header to handle an additional source route; and finally, unless it is specifically stripped at the last-hop router, the option is sent over the (potentially slow) link between the MSR and the MH, thus increasing the traffic in a possibly slow link. It is also impossible to do nested tunneling with source routing, whereas it is possible to encapsulate an already encapsulated packet (a practice not recommended, but still feasible). The overhead in terms of additional network traffic due to larger packets, is higher for encapsulation than it is for source routing (20 octets in the case of IPIP encapsulation, versus 12 octets (option header plus twice four octets for the IP addresses of the source and target MSR, plus padding) in the case of source-routing), but the benefits of using encapsulation outweigh the additional eight octets per packet.

MICP is the protocol used to acquire and distribute information about MHs. The various packet types, as referred to later in the document, are PING, BEACON, GREETING, GRACK, GRNACK, FWDPTR, FWDACK, WHOHAS, IHAVE, OTHERHAS, and POPUP. The exact contents are defined in [9].

Separate IP protocols were necessary (rather than using UDP) both to keep the datagram size down and because all of IPIP and some of MICP processing is done

in the kernel. The exact formats are documented in [9], and their implementation is described later in this paper.

In summary, using MICP for discovery and IPIP for tunneling, the MSRs 'conspire' to 'heal' the partitioned network, and make a collection of heterogeneous network fragments appear to the rest of the network as a seamless subnet.

Route Dissemination

Assume that an organization network has N_R routers, of which N_M are MSRs, and N_m mobile hosts. Let μ be the mobility of MHs, expressed in number of cell switches per MH per unit time. Also, let ν be the average number of hosts, mobile or not, that an MH is communicating with (i.e., receives file service from, has virtual circuit connections to, etc.). Observe that μ is a function of the users' mobility habits, but also of the size of the network, and the size of its cells; a denser higher-speed, network, will tend to have smaller, more confined cells (to support larger numbers of mobiles at higher data rates). The value of μ can vary widely, from 10 switches per second (e.g., a car driving through a densely populated area with very small cells, 'microcells'), to 10^{-4} (switches per second), or less than once an hour (e.g., students moving between classrooms, managers moving between meetings, etc.). The quantity $N_m \times \mu$ gives the number of cell switches per unit time in the network. ν, on the other hand, will tend to be fairly constant and small, probably below 10. Naturally, such numbers are very rough estimates, and are derived by considering what the mobility profile of an average user would be, and how many servers/services that users would be accessing. More experience with mobile data networking is needed before more accurate estimates can be obtained. In the remainder of this section, we shall use $\mu = .001$ (roughly three times an hour), and $\nu = 5$.

Let us now justify the decision to use an embedded network approach, as opposed to simply distributing host routes among the organization's network's routers. If host-routes were to be distributed every time a host moves, as would be the case in a non-embedded-network solution, the load imposed on the network would be proportional to the number of routers involved, the number of mobile hosts, and their mobility. The routing traffic would thus be $T_{hostroutes} = N_R \times N_m \times \mu$, expressed in number of (routing update) messages per unit time. Note that this quantity essentially increases quadratically with the size of the network, as both the number of routers and the number of mobile hosts are a measure of the size of the network.

If host routes are not used, but rather an embedded-network approach is adopted, the problem of how to inform MSRs of where each MH is still remains. There are two extreme solutions: whenever an MH migrates inform all the MSRs, or whenever an MSR needs to know the location of an MH, in queries the other MSRs. Let us examine the traffic imposed on the network by such arrangements. In both cases, tunneling would be necessary to get the packets through, thus increasing the *data* traffic on the network; however, this increase is small (if the average packet size is 500 octets, it is 4%), but more important, it is *predictable* and the resulting traffic increase is smoother and varies linearly with the size of the network (i.e., the number of MHs).

The first case is similar to the host-routes case, except that now only MSRs are involved; presumably, $N_R >> N_M$, (notice that N_R includes N_M) and hence the routing traffic generated, $T_{msrroutes} = N_M \times N_m \times \mu$, is already smaller than $T_{hostroutes}$. This is at the expense of having to encapsulate each message in IPIP.

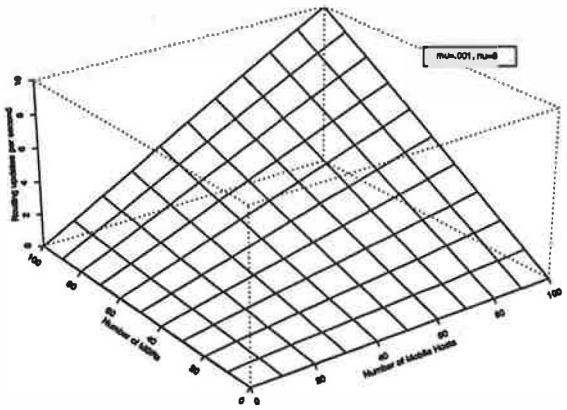


Figure 2: Traffic from distributing routes among MSRs.

The second case is similar to the way ARP[16] works. In ARP, when a host needs to map an IP address to a hardware address, it broadcasts an ARP request on the local network, and the host with the requested IP address responds. Here, an MSR needs the IP address of another MSR handling an MH, and it multicasts a request, asking for the handler. The important points are:

1. The requests only happen when there has not been traffic between the mobile and the host it is trying to communicate with.
2. The results of the queries are cached to minimize requests.
3. The resulting traffic is proportional to the number of other hosts a mobile is communicating with.

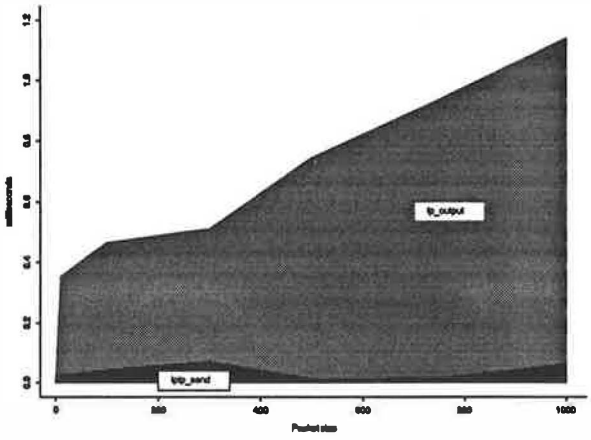


Figure 6: Time spent encapsulating and sending datagrams.

4 Mobile Host Software

An MH that only moves around its own campus needs no kernel modifications, as it always relies on an MSR for service. It needs, however, to handshake with the MSRs and reset the default route whenever it changes cells. The program handling these operations is `mhmicp`; it uses an `IPPROTO_MICP` raw socket to talk to its MSR(s), and thus needs to run as `root`.

Data Structures

An MH keeps two kinds of information:

Configuration information: Its IP address and netmask, the waiting-for-beacon and waiting-for-ack timeouts, and the the number of beacons to be lost before it starts hunting for a new MSR. Also, the name of a script (`huntscript`) to run when it needs to hunt for a new beacon.

State: The following variables define the state of the `mhmicp` process.

- `currentstate`; one of WTG4BEACON, BCNRXED, INCELL, and INCELLPENDINGPING.
- `timestamp`; with a finer temporal resolution than the time needed for cell-to-cell moves.
- `currentmsr`; IP address of the MSR as advertised in the beacon.
- `currentrouter`; source IP address of the beacon, to be used as the default route.
- the pinging timeouts.
- the list of previous MSRs in whose cells the MH has been, but has not yet received an ack.

Algorithms and Implementation

Initially, the `mhmicp` state variable `currentstate` is set to WTG4BEACON. If a beacon is not received within `waiting_for_beacon` seconds, `huntscript` is run to change external parameters of interfaces (e.g., spreading code for a spread-spectrum RF interface), or change interfaces in the case where a unit has multiple interfaces, and the process restarts. If an MICP_BEACON packet is received, the state changes to BCNRXED, the source address of the packet is set as the default router and also placed in the previous-MSRs list, and an MICP_GREET packet is sent to the MSR. In addition, the ARP cache of the MH is flushed. While in this state, all beacons are ignored. If the `waiting_for_ack` timeout expires, the system moves back to the WTG4BEACON state. If an MICP_GRACK packet is received, the process moves to the INCELL state, removes the current MSR from the pending list, and schedules a ping timeout to go off in half the expiration interval supplied with the MICP_GRACK packet. When that timeout goes off, the MH will move to the INCELLPENDINGPING state, send an MICP_GREET to the MSR (at which point the MSR will reset its corresponding `mhinfo` entry to the maximum value again. The MH halves the remaining time and schedules another interrupt, and so on, until it gets an `micp_grack` from the MSR, at which point it resets the timer to its original value, and moves back to the INCELL state.

While in all but the WTG4BEACON state, beacons from other MSRs are simply ignored. If the epoch number in the current MSR's beacon changes, the MH must regreet, as this indicates that the MSR lost its internal state. Also, with each beacon received, the timeout watching for lost beacons is reset. If several consecutive beacons are lost (5 in our implementation), the timeout will go off, add the current MSR to the pending MSRs list, and move the process to the WTG4BEACON STATE. The same holds true if at any point an MICP_GRNACK from the MSR is received. Finally, as long as the previous-MSRs list is non-empty, the MH will keep sending it to the MSR, with a linear backoff bounded by the expiration timeouts of the individual MSRs. This is because, if an MSR is unreachable for more than that period, it will expire the corresponding entry anyway, so there is no reason to keep asking the MSR to send the MICP_FWDPTRs to them.

Performance

The three timeouts are all set to five seconds in our implementation, thus the MH never has to wait more than five seconds if it loses a beacon. The MSRs beacon every second, and advertise an expiration timeout of one minute. This means that the MHs have to process beacons every second, and ping every thirty seconds. On the

average, the `mhmicp` process takes about 0.2% of the CPU time on a 20MHz i386 machine. The `mhmicp` and `pumicp` programs share about 2000 lines of C code, and each one has an extra 600 lines.

While signalling overhead is not a problem with MHs, subtle interactions between periods when the MH is off the network (e.g., in the process of changing cells), or when it first tries to establish a connection, and timeouts in transport- and higher layer protocols (such as TCP or NFS) may be felt by the users. Figure 7 shows the time needed to open a TCP connection to a mobile address; for comparison purposes, a stationary host, two different MSRs (known by their mobile addresses), a local MH and a Popup, are used. The box plot summarizes 100 attempts to connect to the `discard` port (TCP port 9) for each host. Connections to the stationary host complete immediately; connections to mobile addresses take almost 6 seconds; this is because the first SYN packet causes the nearest MSR to send out a WHOHAS request. The originating host waits six seconds before it will send a second SYN, which can now get routed to the mobile address and complete the connection. Observe how a small number (4) of packets to the Mobile must have gotten lost, and it took 12 seconds to complete the connection. This indicates the need for caching a packet that causes a WHOHAS to be transmitted. This, however, would add to the complexity of the MSR code.

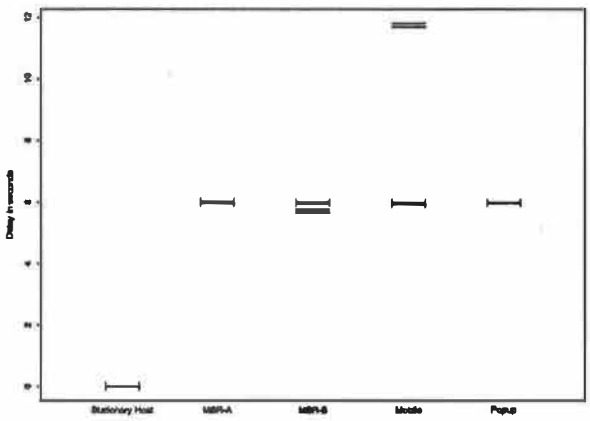


Figure 7: Startup times for TCP.

Another concern[1], is what happens when a mobile has an open TCP connection to another host, which is sending it a continuous stream of data, and the mobile gets temporarily disconnected from the network; the TCP window will fill on the stationary host's side, and as acknowledgements will not be arriving, TCP will back off, and retransmit less and less frequently. When the mobile moves back into the coverage area, it will have to wait for the next retransmission from the stationary host to occur before it can continue receiving its data, and this can take up to a minute. Such behavior can be fixed by 'kick'-ing TCP whenever network connectivity is reestablished. This, however, is bound to be a problem with a lot of high-level protocols that depend on timeouts, and shows that some adjustments may be needed to transport and higher protocols in order for them to work well in the presence of frequent network outages.

[1] This problem was pointed out to us by Andrew Myles (School of Mathematics, Physics, Computing, and Electronics, Macquarie University, Sydney, Australia; *andrewm@mpce.mq.edu.au*), during his visit to Columbia University in August 1992.

5 Popup Software

A Popup is a mobile that has wandered away from its home campus and wants to communicate back. To achieve that while maintaining its home address, it has to acquire a temporary address, called its Nonce Address, from the network it is visiting. This is because the mobile is now in a different administrative domain, whose MSRs, even if they exist, do not communicate with the MSRs in its home campus. Using the Nonce Address, the mobile sends an MICP_POPUP message to one of its home MSRs. The handshake is similar to that of a local mobile, including pings, except that there is no beacon. The MSR uses the Nonce Address as the remote endpoint to tunnel datagrams back to the popup.

Algorithms and Implementation

How the nonce address is acquired is not very important; we supply it manually, but a protocol such as DHCP [3] should be used.

The main problem is how to accommodate two addresses (the home address and the nonce address) on a machine with just one interface, and also do tunneling to and from its home campus (tunneling *from* the popup *to* anywhere else is not strictly necessary, but smart routers may see packets coming in from the wrong interface, and drop them. The easiest solution was to define a Virtual Interface (vif), described next. When a popup shows up on a foreign network, it acquires a nonce address which it assigns to its real interface, then assigns its home address to the virtual interface, and then routes all packets not explicitly destined for the home MSR through the virtual interface.

Virtual Interface

The virtual interface (`vif`), much like the loopback interface, has no input queue. Packets sent to it by `ip_output()` are looped back if their destination

address is the same as `vif`'s; otherwise, they are encapsulated in IPIP and tunneled to the home MSR using the real interface. All packets are routed through the virtual interface, by doing the equivalent of:

```
/etc/route add default home-address 0
```

that is, adding a default route that is the interface itself. Thus (see Figure 8), outgoing traffic is routed through the virtual interface tunneled to the home campus, and routed from there on. A packet *for* the popup is first routed to its home network, where an MSR picks it up, tunnels it to the MSR handling the popup, if necessary, which in turn tunnels it again back to the popup. The packet received has protocol type IPIP, so it is sent to the input routine of IPIP, which strips the header and feeds the packet back into the IP output queue. But now this is a packet for the mobile address of the popup, and it is simply looped back and delivered to the appropriate transport protocol.

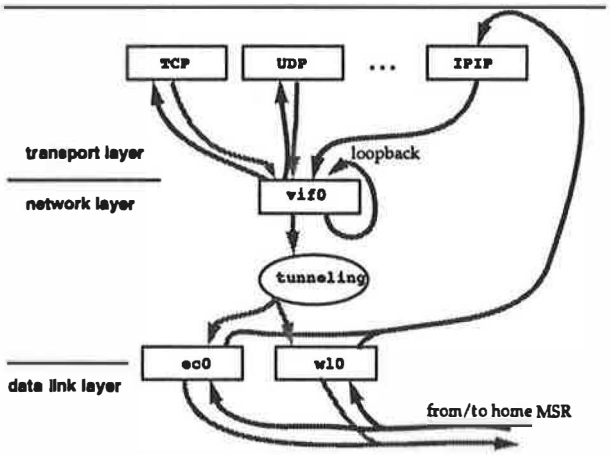


Figure 8: VIF processing.

Performance

Since there is no constant beacon, the control process, `pumicp` takes even less CPU time, about 0.1% of the CPU on a 20MHz 386 machine. However, the popup has to do its own tunneling, which adds 100 μs to each outgoing packet (this is a slower machine than the 486/33Mhz showing the 45μs encapsulation time). Since, however, network interfaces on mobiles are not expected to run at the full bandwidth, the encapsulation delay merely adds to the latency, which is probably insignificant compared to the latency of traversing a wide area network.

The code for the vif driver (`netinet/if_vif.c` and `netinet/if_vif.h`) is under 300 lines of C code. The interface is `if_attach()`ed when a special device file (`/dev/vif`) is opened, which also allows the use of `ioctl()`s to set the address of the home MSR for use by the tunneling code.

6 Summary

We have presented an infrastructure which enables mobile machines to keep their network connections while they move in a networked environment. A lot of fine detail, such as packet formats, has not been covered here, as it can be found in the official specification [9]. The emphasis was on describing and justifying the major design decisions, and describing the reference implementation and its performance.

To summarize, our design for mobility follows the embedded network model, whereby Mobile Hosts keep their IP address as they migrate, and the addresses of all MHs in an organization's network belong to the same logical subnet, the Mobile Subnet. Special routers, the Mobile Support Routers, keep track of the MHs' location, in order to be able to route (tunnel) packets to and from them. By allowing Popup MHs to acquire secondary addresses, and use tunneling to communicate to their home MSRs, we extend the ability to be mobile to outside the limits of an organization's network, or even to parts of that network without MSRs. The ability to use MSRs the way we are using them should be viewed as an optimization of the general case. In order to keep the routing updates traffic low and scalable, we use on-demand discovery of MH locations, rather than gratuitously propagate their location to all the MSRs in a network.

Finally, we present the performance of the reference implementation; the figures show that a medium-power machine, such as a 486/33Mhz "clone" can more than adequately perform all the tasks of an MSR: MH registration and tracking, exchange of routing information with the other MSRs as well as the regular routers, and route/tunnel packets between MHs in its cell(s) and the hosts they communicate with. As far as the impact of mobility on the MHs' performance is concerned, the signalling necessary is negligible, although the mechanics of on-demand acquisition of routes adds delays in the setup of connections.

Acknowledgements

This work was supported in part by National Science Foundation grant CDA-9022123, IBM Corporation, and the Center for Telecommunications Research, an NSF Engineering Research Center funded by grant ECD-88-11111. It has benefitted from discussions, comments, and suggestions by a lot of people, including Dan Duchamp, Steve Deering, Mike O'Dell, Phil Karn, and the Mobile Hosts Working group of the IETF.

Bibliography

[1] Danny Cohen, Jonathan B. Postel, and Raphael Rom. IP Addressing and Routing in a Local Wireless Network, July 1991.

[2] D. C. Cox. Personal Communications — A Viewpoint. *IEEE Communications Magazine*, pages 8–20 and 92, November 1990.

[3] R. Droms. Dynamic Host Configuration Protocol. Internet-Draft, available as `draft-ietf-dhc-protocol-04.txt`, August 1992.

[4] Sam Ginn. Personal Communication Services: Expanding the Freedom to Communicate. *IEEE Communications Magazine*, pages 30–39, February 1991.

[5] C. L. Hedrick. Routing Information Protocol. RFC 1058, June 1988.

• [6] Charles L. Hedrick. An Introduction to IGRP, August 1991. Available with anonymous FTP from `ftp.cisco.com`, file `igrp.doc`.

[7] SRI International. Network reconstitution protocol. Technical Report RADC-TR-87-38, Rome Air Development Center, June 1987.

[8] John Ioannidis, Dan Duchamp, and Gerald Q. Maguire Jr. IP-Based protocols for mobile internetworking. In *Proceedings of SIGCOMM'91*, pages 235–245. ACM, September 1991.

[9] John Ioannidis, Dan Duchamp, Gerald Q. Maguire Jr., and Steve Deering. Protocols for Supporting Mobile IP Hosts. Internet-Draft, June 1992.

[10] John Jubin and Janet D. Tornow. The DARPA Packet Radio Network Protocols. *Proceedings of the IEEE*, 75(1):21–32, January 1987.

[11] William C. Y. Lee. *Mobile Cellular Telecommunications Systems*. McGraw-Hill, 1989.

[12] Barry M. Leiner, Donald L. Nielson, and Fouad A. Tobagi. Issues in Packet Radio Network Design. *Proceedings of the IEEE*, 75(1):6–20, January 1987.

[13] Richard J. Lynch. PCN: Son of Cellular? The Challenges of Providing PCN Service. *IEEE Communications Magazine*, pages 56–57, February 1991.

[14] D. L. Mills. DCN Local-Network Protocols. RFC 891, December 1983.

[15] Radia Perlman. *Interconnections: Bridges and Routers*. Addison-Wesley, 1992.

[16] D. C. Plummer. Ethernet Address Resolution Protocol. RFC 826, November 1982.

[17] Jon Postel. Internet Protocol. RFC 791, September 1981.

[18] Jon Postel. Transmission Control Protocol. RFC 793, September 1981.

[19] Joyce Reynolds and Jon Postel. Assigned Numbers. RFC 1340, July 1992.

[20] Ian M. Ross. Wireless Network Directions. *IEEE Communications Magazine*, pages 40–42, February 1991.

[21] Richard M. Singer and David A. Irwin. Personal Communications Services: The Next Technological Revolution. *IEEE Communications Magazine*, pages 62–66, February 1991.

[22] Douglas G. Smith. Spread Spectrum for Wireless Phone Systems: The Subtle Interplay between Technology and Regulation. *IEEE Communications Magazine*, pages 44–46, February 1991.

[23] Raymond Steele. Deploying Personal Communication Networks. *IEEE Communications Magazine*, pages 12–15, September 1990.

[24] C. Sunshine and J. Postel. Addressing Mobile Hosts in the ARPA Internet Environment. IEN 135, March 1980.

Author Information

John Ioannidis is a last-year graduate student in the Computer Science department at Columbia University, and by the time you read this he should have received his PhD and should be working at Bellcore. In addition to his thesis work on Mobile Internetworking, JI has interests in all aspects of System Design, including networks, operating systems, and security. Reach him electronically at *ji@cs.columbia.edu* .

Gerald (Chip) Q. Maguire Jr. is associate professor of Computer Science at Columbia University in the City of New York. His research interests include distributed computing, wireless networking, building portable software systems and special-purpose processors, picture archiving and communication systems (PACS), and image processing/computer graphics with an emphasis on medical images. Reach him electronically at *maguire@cs.columbia.edu* .

Reach both authors via paper mail at the Department of Computer Science, Columbia University, New York, NY 10027.

Mobile Computing Environment Based on Internet Packet Forwarding

Hiromi Wada, Takashi Yozawa, Tatsuya Ohnishi, & Yasunori Tanaka
– Matsushita Electric Industrial Co., Ltd.

ABSTRACT

We have explored a mobile computing environment which provides migration transparency of portable hosts. In this paper, we propose a means of continuous communication with mobile hosts called the "Packet Forwarding Method (PFM)". In the environment, each mobile host has a home address, and when it migrates to another network, it is also assigned a temporary address. An application on the mobile host always uses the home address for communication.

PFM is based on packet forwarding. A packet destined for a home address of a mobile host is forwarded to its current temporary address. This forwarding is performed by a "Packet Forwarding Server (PFS)" or by the sender host itself internally.

This method has adaptability for existing multi-vendor environments since enhancement of stationary hosts is optional and modification of routers is not required. Stationary hosts which have been enhanced to have forwarding functionality can communicate with mobile hosts more efficiently than those without the enhancement.

The implemented prototype code size is relatively small, and experiments indicate that communication overhead is trivial, especially in the case of stationary hosts with the enhancement.

Introduction

Recently, the physical portability of workstations has been improved as they have become smaller and lighter, while software on workstations has grown to achieve higher functionality and increased in scale. These promote workstations' dependency on servers for sharing software, data, and physical resources, such as printers and mass storage devices. Communication facilities such as electronic mail systems and electronic bulletin boards have also been enhanced and improved. In the near future, wireless networks will spread even more widely, and continuous communication unaffected by hosts' migration will be an essential requirement. Therefore, for optimal workstation use, it is desirable to provide continuous communication with hosts whose physical location changes.

In the usual Internet, however, IP addresses are not only identifier but also the hosts' location information. Therefore migrating hosts have to change their IP addresses in order to be connected to a new network. As a result, the migrating hosts are no longer able to communicate with the other hosts using their old addresses. This is the fundamental problem that prevents migration transparency. One of the ideal solutions is to replace the current Internet architecture with a new one which provides migration transparency. However, it seems to be too costly for existing environments, therefore it does not seem to be a practical solution. We believe that it is more important to provide compatibility with current Internet and multi-vendor environments.

Goals

In this section we present our goals of the study briefly. We have explored a computing environment where hosts can communicate with each other continuously when they migrate across networks, in a fashion that is transparent to layers above IP. Our primary goal is to establish a method to realize such a mobile environment. We think that this goal can be broken down into developing the two basic mechanisms below.

- **Routing mechanism** Dynamic routing of packets destined for a mobile host is the most fundamental function for the environment. The routing should work transparently to application entities in the environment.
- **Location information management mechanism** Location information of mobile hosts should be delivered to hosts so that they could perform the routing described above.
  The following conditions should be satisfied to make our approach to be effective for various environments.

Regarding implementation, the impact on existing applications should be minimized, and backward compatibility with existing environments should not be damaged. Namely,

- No changes to routers.
- No changes to mobile hosts above the network layer.
- No mandatory changes to the networking software of stationary hosts.

For performance to be practical level, the following point seems to be significant:

- **Minimize the cost of host tracking** To deliver packets to a mobile host correctly, up-to-date location information of the host should be known to a certain set of hosts. The cost for distributing such information should be minimal.
- **Optimize the forwarding route** Obviously, inefficient forwarding route causes a serious performance problem. It is required that forwarding routes is optimized dynamically to adapt to change of location of an MH.

Basic Concept

We assume three active entities in our scheme: a Mobile Host (MH), a Packet Forwarding Server (PFS), and a Stationary Host (SH). Each host is regarded as one of these entities in our scheme. Transparent migration is implemented using a protocol we call the "Internet Packet Transmission Protocol(IPTP)". IPTP is used for packet forwarding and to maintain MH location information.

The basic concept of our scheme is as follows. Packets are sent to an MH specifying its "home" address. It is the "home" address that an application entity sees as an ordinary IP address. When the MH migrates to another subnet, it is given a temporary IP address on that network. A packet sent to the home address is intercepted by a Packet Forwarding Server(PFS) process that remains behind on the original network. The PFS is responsible for tracking the location of the mobile host, and hence knows the temporary IP address of the MH. Intercepted packets are encapsulated and forwarded specifying the MH's temporary address.

Our mechanism provides two operational communication modes depending on whether the sending host can be modified or not: autonomous mode and forwarding mode. In autonomous mode, the packet forwarding is performed by the networking software of the sending host. The home address of the MH in the destination address field of the packet is overwritten by the temporary address currently being used by the MH. In forwarding mode, packets destined for the home address of the MH are picked up by the PFS on the MH's home network and forwarded to its current temporary address.

Autonomous mode allows two hosts to communicate via normal internet routing. Although location information is maintained with the help of the home PFS, packet encapsulation is done by the sender. None of the intermediate hosts such as routers need know anything about mobility management. However, autonomous mode requires modifications to the sending host.

Forwarding mode allows an unmodified host to communicate with arbitrary MHs. Packets sent to the MH are transmitted via standard IP protocol but received by the PFS unbeknownst to the sender. Using IPTP, the PFS then forwards the packets, completely insulating the original sender from coping with mobility.

Terminology

This section defines terms and conventions used throughout this paper.

- **home MH** (for a network or a PFS): An MH whose home is in the network (where the PFS lives).
- **home PFS** (for an MH): A PFS which lives in the home network of the MH.
- **home address** (of an MH): An address assigned to the MH, which does not vary even in MH migration. See temporary address.
- **home network** (of an MH): The network of the home address of an MH.
- **mobile host** (MH): A host which can migrate across network boundaries. See stationary host.
- **path**: a set of routes between two particular hosts. A path is available when at least one route of the path is available.
- **stationary host** (SH): In the original meaning a host which does not migrate across network boundaries. In this paper, however, it is used as a representative term of a peer host communicating with a mobile host. Of course, that is not necessarily stationary.
- **temporary address** (of an MH): An address assigned to an MH which varies with MH migration reflecting the current location. The network part of the temporary address is always equal to the current network address. A temporary address is thought to be dynamically assigned to an MH when it visits another network. See home address.
- **visited PFS** (by an MH): A PFS which lives in a network where the MH visited.
- **visiting MH, visitor MH**: An MH which visited the network other than home network.

Addressing

In this section, we describe how mobile hosts are located, identified and assigned their addresses. In the mobile computing environment that we propose, each MH is assigned an address on one particular subnet that is distinguished as its "home" network. The MH retains its home address regardless of migration. The network part of the home address is equal to the home network address. Usually, a home address is not changed. A PFS on the MH's home network is responsible for forwarding packets destined for the MH.

In contrast, a temporary address reflects the current real location of the MH (we also refer to it as the "real address"). The network part of the

address is equal to the network address where the MH currently lives. A temporary address changes every time the host migrates to another network. The assignment process of temporary addresses is beyond the scope of this paper.

Applications address an MH using its home address, regardless of its location. This is true both for applications running on the MH and for applications on other machines that must communicate with the MH. To send data to an MH, a host builds an IP packet whose destination address is the home address of the MH. These packets are then encapsulated in special packets whose destination address is the current temporary address of the MH. Packets are then delivered to the MH via normal internet routing. An MH receiving an encapsulated packet decapsulates it to the original IP packet whose destination address is equal to its home address.

The temporary address of an MH is assigned dynamically when the MH visits a network. As a result, encapsulated packets might mistakenly be sent to a temporary address which has been reassigned to another MH. To filter out packets mistakenly received, each encapsulated packet is tagged with the home address of the destination MH. Since each MH has a unique home address, it is possible to distinguish packets that should not actually be delivered. An MH considers an encapsulated packet as destined for itself only if the destination address and home address of the packet are equal to its temporary address and home address.

Packet Forwarding Method(PFM)

As formerly described, Packet Forwarding Method consists of two mechanisms. One is a packet forwarding mechanism, and the other is a location management mechanism.

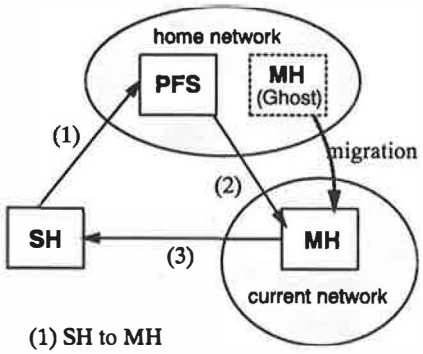
Packet Forwarding Mechanism

In this section, we describe how packet forwarding works in detail. As described in the above section "Basic concept", packet forwarding is performed in one of two modes: forwarding mode and autonomous mode. In forwarding mode, packet forwarding is performed by home PFSs only. In autonomous mode, packet forwarding is mainly performed by sending hosts themselves, though home PFSs and visited PFSs (autonomous supporter PFSs) also do as well. Both modes can co-exist in one environment.

In the following section we describe the forwarding mode, and then in the next section the autonomous mode.

The Forwarding Mode

In this mode, packet forwarding for an MH is performed by a PFS on the MH's home network. Figure 1 illustrates this case.



(1) SH to MH
(2) Packet Forwarding
(3) MH to SH

Figure 1: Forwarding mode by a home PFS

In the figure we consider communication between an SH and an MH.

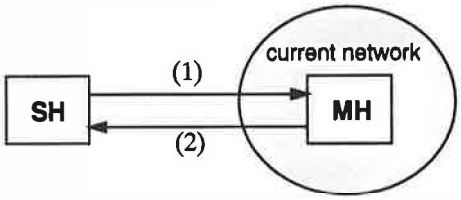
1) SH to MH – An SH sends a packet to an MH specifying the MH's home address. The packet is routed to the MH's home network where it is intercepted by a PFS. Packet interception is arranged by the PFS either by using some sort of promiscuous mode or by arranging with local gateways for packets to be routed to the SH on which the PFS is running.
2) Packet Forwarding – The PFS encapsulates the packet sent in (1) into a "Packet Transmission" message (the exact format is described in the section "Packet Format"). The destination address in the IPTP message is the temporary address of the MH. The PFS maintains a mapping between the home address and the temporary address by the IPTP protocol. This location information management is described later.
   When the MH receives the "Packet Transmission" message, its IPTP layer decapsulates the message. From the perspective of applications running on the MH, the MH appears to still reside on its home network.
3) MH to SH – The packet decapsulated out of the "Packet Transmission" message contains an IP address of the SH as the source address field. The MH sends a reply packet directly to the SH by normal IP protocol without any assistance of IPTP layer of the SH or a PFS. The source address in the packet is the MH's home address. Hence the reply packets are routed according to the normal IP routing mechanism. We assume that the MH dynamically acquires even static routing information through a protocol such as DHCP [8].

The Autonomous Mode

Forwarding by an SH

In this mode, packet forwarding is mainly performed directly by the sending host, instead of by the PFS. Figure 2 illustrates this case. The SH

maintains a mapping between the home address and the temporary address of the MH. In this respect the functionality added to the networking software of the SH is essentially equivalent to that of the PFS, though there are of course significant differences between them.



(1) SH to MH
(2) MH to SH

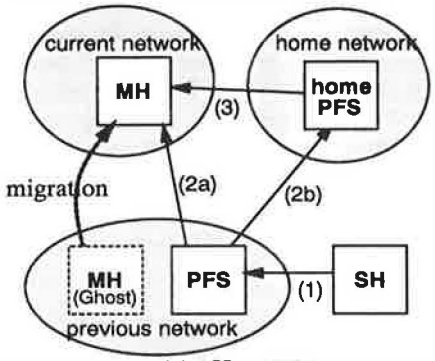
Figure 2: Autonomous mode

1) SH to MH – The SH sends a packet directly to the MH. Applications on the SH use the MH's home address. IPTP software on the SH encapsulates the packet into an "Packet Transmission" message and send it to the MH using the MH's current temporary address. On receiving the packet, the MH's IPTP software decapsulates the original IP packet and passes it to the appropriate protocol.
2) MH to SH – The MH sends a packet directly to the SH. This is the same as in the forwarding mode above.

Forwarding by an autonomous supporter PFS and a home PFS

This can happen when a packet is destined for a visitor MH which has gone to another network. An autonomous supporter PFS forwards a packet destined for such visitor MH. If the autonomous supporter PFS knows the current temporary address of the visitor MH, it forwards packets destined for the obsolete temporary address in the network to the current temporary address. Otherwise, it forwards them to the home address of the visitor MH and then the home PFS forwards them to current temporary address of the MH. The autonomous supporter PFS obtains the address of the home PFS from the IPTP header in the received packet. Figure 3 illustrates this case.

1) SH to MH – The SH in autonomous mode sends a packet directly to the temporary address of the MH, although the MH does not use the temporary address any longer.
2) Packet Forwarding – If the autonomous supporter PFS knows the current temporary address of the MH by the mechanism described in the section "Location Information Management", it forwards the packet to the current temporary address(2a). If not, it forwards the packet to the home address(2b). When forwarding, the PFS does not encapsulate the packet because it is already encapsulated. Instead, it decrements the counter field in the IPTP header by 1 and if the value is 0, the PFS discards the packet. The autonomous supporter PFS basically inspects all IPTP messages on its network.
3) Packet Forwarding – This takes place only when preceded by (2b). The home PFS forwards the packet to the current temporary address of the MH. Forwarding process is the same as (2) above (counter decrement and no encapsulation).



(1) SH to MH
(2) Packet Forwarding
(3) Packet Forwarding

Figure 3: Forwarding mode by an autonomous supporter PFS

Location Information Management

In this section we describe how location information is maintained with the IPTP. Up to date location information is required for packet encapsulation by PFSs and SHs running in autonomous mode. Location information for an MH is transmitted from the MH itself to its home PFS. The home PFS is then responsible for propagating the data to all concerned hosts, though in a particular situation an autonomous supporter PFS propagates the data for efficiency. Below, we describe the way in which this information is distributed.

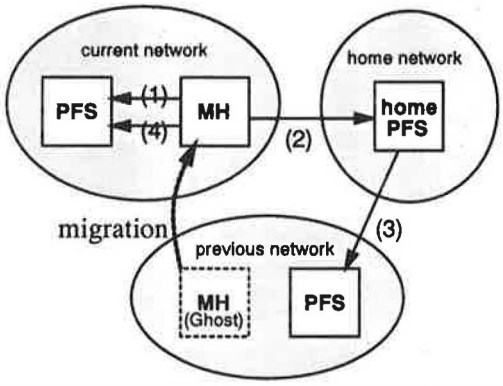
MH Migration

When an MH moves from a network to another network, location information is distributed as shown in Figure 4.

0) The MH is assigned a temporary address.
1) Ping Autonomous Supporter – The MH tries to find a PFS which can support autonomous mode in the new network. This is done by broadcasting a "Ping Autonomous Supporter" message and seeing if any PFS responds. If a reply is found, the MH can transmit packets in autonomous mode. When a PFS receives a "Ping Autonomous Supporter", it sends a reply message to the sender if it supports the mode, otherwise, it silently discards the message.
2) MH Location Information – The MH sends an

"MH Location Information" message to a PFS on its home network. This message carries the home and temporary addresses of the MH, an autonomous flag which indicates whether the MH can communicate in autonomous mode or not, and the address of the PFS which responded to the "Ping Autonomous Supporter" message.

When a PFS in the home network receives an "MH Location Information" message, it returns an acknowledgment to the MH, and updates its mapping for the home address of the MH.



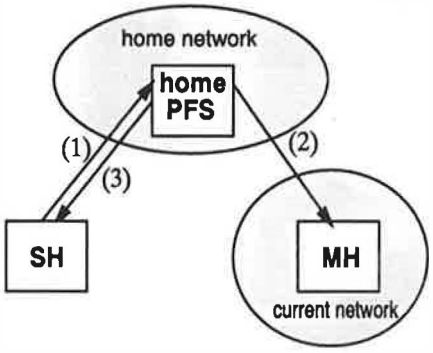
(1) Ping Autonomous Supporter
(2) MH Location Information
(3) MH Location Information
(4) MH Visiting

Figure 4: MH Migration

3) MH Location Information – The home PFS now determines whether the MH has just migrated from its home network. If not then the MH has moved from one remote network to another. If the MH was operating in autonomous mode on the previous remote network, the home PFS sends an "MH Location Information" message to the PFS on the previous remote network. When the previous remote PFS receives the "MH Location Information" message, it acknowledges the message and flags its mapping (if any) for the MH specified in the message. The flag indicates that the MH has migrated, and means that the PFS may delete the MH's entry if necessary.
4) MH Visiting – After the MH receives an acknowledgment of its "MH Location Information" message from its home PFS, it sends an "MH Visiting" message to the PFS which responded in (1). The message includes the home and temporary addresses of the MH. The PFS which receives the "MH Visiting" message registers the MH in its visitor MH list.

Packet to Home Address of Migrated MH

A host that communicates with an MH that has migrated will address its packet to the home address of the MH. If the sending host supports autonomous mode, then its location tables must be updated to reflect the new location of the MH. Figure 5 depicts how this update procedure takes place.



(1) SH to MH
(2) Packet Forwarding
(3) MH Location Information

Figure 5: Packet to Home Address of Migrated MH

1) SH to MH – The SH sends a normal IP packet to the MH's home address.
2) Packet Forwarding – The home PFS picks up the packet, encapsulates it within an IPTP "Packet Transmission" message, and sends it to the MH's current temporary address.
3) MH Location Information – If the MH has moved to a network that supports autonomous mode then the home PFS attempts to notify the SH that autonomous mode communication is possible. If the SH is capable of autonomous mode communication, when it receives the "MH Location Information" message, it caches the MH's new temporary address and enters autonomous mode for all packets destined for the MH.
4) The SH can now send packets to the MH without an intervening hop through the PFS.

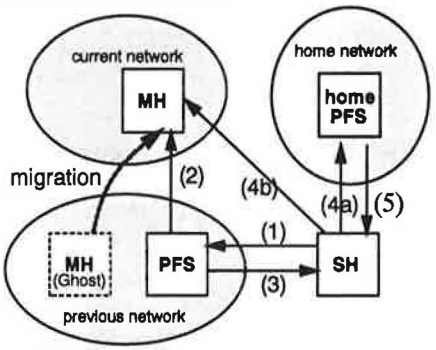
Packet to Obsolete Temporary Address

An autonomous supporter is a PFS that provides service to a visiting MH. After the visiting MH departs, packets addressed to it may still arrive either because the sending host is in autonomous mode or the forwarding PFS has not received an "MH Location Information" message yet. The second case is handled by the "MH Location Information" message described in the section "MH Migration". The first case, however, requires additional messages.

Figure 6 illustrates what happens if a PFS acting as an autonomous supporter receives a packet destined for an MH that has already left its network. The PFS must encapsulate and forward the packet. More importantly, however, it must notify the

sending host of the MH's new address. This notification provides lazy updates and route compression as well.

1) SH to MH – The SH sends a "Packet Transmission" message to the MH's old temporary address. The autonomous supporter for the MH's old temporary address will intercept the packet. It knows that the MH is gone because it received an "MH Location Information" as described in the section "MH Migration" (3).



(1) SH to MH
(2) Packet Forwarding
(3) MH Location Information
(4) Subsequent SH to MH
(5) MH Location Information

Figure 6: Packet to Obsolete Temporary Address

2) Packet Forwarding – The PFS may attempt to forward the packet to the MH if the MH's new temporary address is still in its cache. The PFS is not required to maintain the new address of the MH. If it is not on the cache, the PFS knows merely the fact that it has gone. If so, the packet is instead forwarded to the home PFS for correct rerouting.
3) MH Location Information – To implement lazy notification of the SH, the PFS must now notify the SH that it has an out of date address for the MH. It therefore sends to the SH an "MH Location Information" message. The PFS includes the MH's new address if possible, allowing subsequent packets to be routed directly to the MH without going through the home PFS. If the PFS does not know the MH's new address then the address field is set to NULL, indicating that the SH should route packets to the MH's home network where they are picked up by the home PFS.
4) Subsequent Packet Transmission(SH to MH) – The SH knows that its mapping for the MH is stale. If no new address for the MH was received in (3), then it will transmit subsequent packets to the MH's home network for processing by the home PFS (4a). Otherwise, it will transmit subsequent packets to the new temporary address of the MH (4b).
5) MH Location Information – When the home PFS receives the next packet destined for the home address of the MH, if the MH is available for autonomous mode communication, the home PFS sends an "MH Location Information" message to the sender. At the same time, it forwards the packet to the current temporary address of the MH. When the sender receives the "MH Location Information" message, it updates the cache entry of the MH's location and enters autonomous mode.

Internet Packet Transmission Protocol (IPTP)

In this section we describe the Internet Packets Transmission Protocol(IPTP). IPTP consists of five packet types that fit within one packet format. The packet types provide packet forwarding, new address notification, pinging for an autonomous supporter, visiting MH notification, and an IPTP echo check. These various packets are used by PFSs, MHs, and SHs that support autonomous mode.

Message Type

In this section, we describe the different IPTP message types.

1) **Packet Transmission message** This message is used to forward packets from a PFS to an MH and to send packets from an SH to an MH.
2) **MH Location Information message** This message is used to notify a PFS or an SH of an MH's new address. NULL address means that the sender of this message does not know the MH's new address.
3) **Ping Autonomous Supporter message** This message is used by an MH to find a PFS which supports the autonomous mode in a new temporary network. If a PFS that receives this message serves the autonomous mode, it responds to the message. If not, it is not necessary to send any response messages. Therefore, the MH sending this message can perform the same process regardless of whether a PFS, which does not support the autonomous mode, exists or not in the new temporary network.
4) **MH Visiting message** This message is used to signal a PFS which supports the autonomous mode that an MH has come to this network.
5) **Echo message** This message is for examining whether a host employs IPTP or not (the exact usage is not defined).

Packet format

Figure 7 illustrates IPTP packet format. Each parameter means:

1) **Type** This field indicates a message type of an IPTP packet; Table 1 displays the types.
2) **Aim** This field indicates whether the packet contains either a request or a response. Each request message except "Packets Transmission" message requires a response.
3) **Sequence number** The sequence number of the packet transmitted between a requester and a responder.

Value	Meaning
0	Packet Transmission message
1	MH Location Information message
2	Ping Autonomous Supporter message
3	MH Visiting message
4	Echo message

Table 1: Message types

4) **Autonomous** This field is used only in "MH Location Information" messages from an MH to the home PFS. It indicates whether the home PFS should notify SHs of the MH's temporary address or not. Table 2 summarizes the meanings.

Value	Meaning
0	The home PFS must not notify SHs of the MH's temporary address
1	The home PFS may notify SHs of the MH's temporary address

Table 2: Values for 'autonomous'

5) **Counter** The counter is used to detect forwarding loops. It is set to an implementation-specific number whenever a "Packet Transmission" message originates. When a PFS receives the "Packet Transmission" message, the PFS decrements the counter by 1. If the counter is equal to 0, the PFS discards the packet instead of forwarding it.
6) **Status** The status of the packet (e.g., error code).
7) **Home address of MH** The address of an MH on its home network.
8) **Temporary address of MH** The address of an MH on a temporary network. Assigned using some dynamic configuration protocol such as DHCP [8].
9) **Address of PFS** The IP address of a PFS.
10) **Authentication information** A password or token that PFSs use to decide whether an MH has sufficient credentials to be given service. The exact nature of this is beyond the scope of this paper.
11) **Encapsulated packet** This is an original IP packet destined for an MH. This field is only included in "Packet Transmission" messages.

Parameters

Two important transmission parameters for IPTP are the timeout interval and the retransmission count. The timeout interval is the length of time IPTP will wait before retransmitting a packet. The retransmission count is the number of times a packet will be resent. IPTP defines these parameters for all packets except "Packet Transmission" messages. Because IP makes no guarantee about message delivery, IPTP "Packet Transmission" messages can also be lost. Reliable packet delivery is left to higher level protocols in the transport and network layers. For the other message types, we assume that the timeout interval will be tuned to specific implementations. The remaining issue, therefore, is the number of retransmissions.

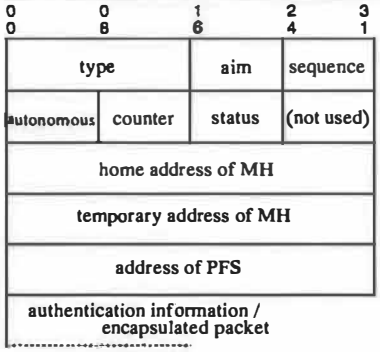


Figure 7: IPTP packet format

Although the exact number of retransmissions should be implementation specific, the cardinality is important. The "MH Location Information" message should be retransmitted an infinite number of times. If for any reason, such as a network failure, an MH cannot notify its home PFS of its new address, the MH will become temporarily lost. If communication between the MH and PFS is ever possible again, then a packet will eventually get through, allowing hosts communicating with the MH to reestablish contact through the home PFS.

The other packet types, "Ping Autonomous Supporter", "MH Visiting", and "Echo", can be retransmitted a finite number of times. Loss of these packets may result in a less efficient routing, but will not be fatal. For instance, a host capable of autonomous mode communication may mistakenly use forwarding mode if a "Ping" message is lost.

Implementation

The first prototype implementation of this method exists under SunOS 4.1.1. It consists of approximately 560 effective lines of user-level code for PFS, 450 effective lines of kernel code and 760

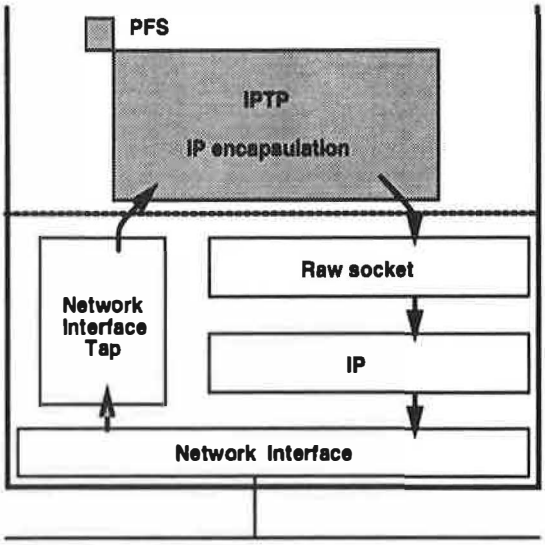


Figure 8: Internal structure of a PFS

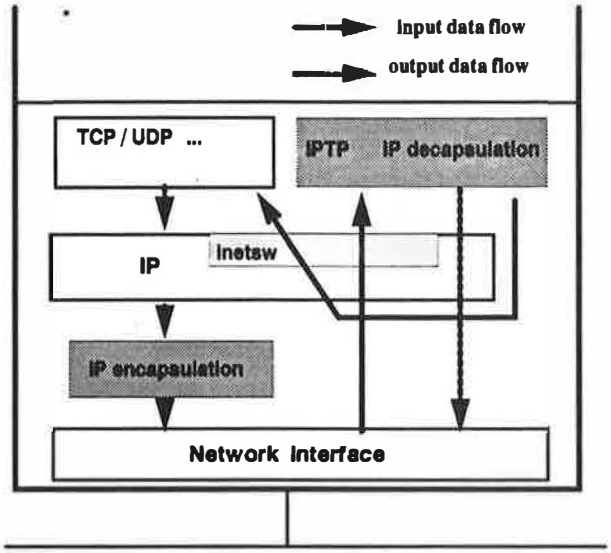


Figure 9: Internal structure of an MH

effective lines of user-level code for an MH and 260 effective lines of kernel code for an SH. It includes most of forwarding mechanism and rudimentary location information management mechanism. Figures 8 and 9 illustrate the internal structures of a PFS implementation and an MH implementation respectively. The internal structure of an SH's implementation is the same as an MH's, though there are some differences in its internal code and function.

In the PFS implementation, an IPTP module is in user space. It waits for IPTP messages with IPTP protocol number passing through Network Interface Tap(NIT) [17]. When it receives an IPTP message, it processes it according to the protocol specification and replies via the raw IP socket. At the same time, it intercepts packets destined for its home MHs which are currently away. When it picks up a packet destined for a home MH via NIT, it encapsulates the packet within a "Packet Transmission" message and forwards it to the MH's current temporary address.

In the MH implementation, an IPTP module is just above the IP module in kernel space. It receives IPTP messages via its inetsw[] table. The inetsw[] table includes an entry for the IPTP. When the IPTP module receives an IPTP message except a "Packet Transmission" message, it sends an response message via the IP module in kernel. When it receives a "Packet Transmission" message, it decapsulates the message and puts the decapsulated IP packet into the IP input queue. When the MH sends a packet to an SH, the packet is processed by normal IP. When the MH sends a packet to an MH, the packet is encapsulated by another IPTP module between the IP module and network interface drivers. This encapsulation module is just the same as an SH has.

The SH implementation also has an entry for the IPTP in its inetsw[] table. SHs exchange IPTP messages in the same way as MHs. The SH implementation differs from the MH implementation in including no IPTP decapsulation facility.

Though not implemented in this prototype, a routing module of an MH is required to be modified. By the modification, the routing module shows the TCP/IP protocol modules the MH's home address as an internet address related to the network interface, while the network interface driver deals with the MH's current temporary address. The current implementation suffices it to hack with ifconfig command.

We expect that the total amount of the IPTP implementation will be small. And the current prototype implementation does not require TCP/IP original code to be changed. Therefore, we think implementation of the IPTP has adaptability to current TCP/IP implementations.

Detailed Discussions

Forwarding Issues

PFS Packet Interception

In order to correctly forward packets for mobile hosts, a PFS must be able to intercept packets addressed to hosts that have migrated away from the local network. One possible implementation is to use a promiscuous mode, if the underlying interface supports it. Such a solution, however, may impose a substantial load as the PFS is forced to inspect every packet.

A more attractive alternative is to use the proxy ARP. When a PFS receives an "MH Location Information" message from an MH, it broadcasts an ARP reply packet for the MH's home address. The reply packet specifies that the MH's IP address now resolves to the address of the PFS's physical

interface. Subsequent packets addressed to the MH's home address will be received by the PFS.

If a PFS is already forwarding packets for an MH, it responds as a proxy to any ARP requests for the MH. The ARP reply message indicates that packets destined for the home (IP) address of the MH should be physically (i.e., at the hardware address level) addressed to the PFS.

Unfortunately, this technique cannot be applied to a PFS acting as an autonomous supporter for a visiting MH. A visiting MH will use a temporary address. This address will eventually be reused when the visiting MH migrates to another network. If the PFS issues a Proxy ARP for this address, packets intended for the new user of the address might lose packets or might end up with unwanted packets. Temporary addresses must be reusable. The consequence is that a PFS may only act as an autonomous supporter if it has a promiscuous interface on broadcast medium that allows it to see all network traffic.

Detection of Forwarding Loops

If an MH is roaming among temporary networks where PFSs support autonomous mode, it is possible that forwarding relays will occur. To prevent a forwarding loop, the "Packet Forwarding" message contains a special counter. When a PFS forwards packets at first, it sets the counter to an upper bound defined by the system. Before another PFS forwards the packet, it decrements the counter by 1 and it compares the value to zero. If the PFS finds the counter equal to zero, the packet is discarded. Otherwise the packet is forwarded normally.

Multiple Forwarding

If only a home PFS forwards packets destined for home MHs, the communication route between MHs and SHs will be simple. However we also want that an autonomous supporter PFS forwards packets destined for obsolete temporary addresses of visitor MHs who have gone to other networks. We believe that any packets destined for MHs should not be lost except during the period they are completely disconnected from all networks. This is the reason why we choose multiple forwarding.

Dealing with broadcast packets

Considering the nature of broadcast, broadcast packets are inconsistent with mobile computing environments. We think it is not necessary for broadcast packets to be forwarded to MHs on other networks. Broadcast packets should be dealt only within a physical network.

In our method, broadcast packets in a home network are not forwarded to MHs which have migrated to other networks. However, broadcast packets in a current network can be received by an MH, and sent by specifying the broadcast address of the current network.

Information Management Issues

Translation Tables

Address Translation tables are maintained for PFSs, and SHs and MHs using autonomous mode. A table entry contains the home address and current temporary address for an MH. In addition, each PFS table entry that represents a home MH will contain the address of the PFS in the MH's current network. An SH table entry will contain the address of the MH's home PFS. Translation tables are maintained on MHs in exactly the same way as SHs.

PFSs are responsible for providing non-volatile storage for translation information. SH and MH tables are only caches for data managed by PFSs. An SH or an MH can easily refresh its tables by interacting with the appropriate PFS. Of course, a disastrous failure might cause a PFS to lose its translation information. If this occurs, the information must be recovered by inducing MHs to resend "MH Location Information" messages. This might have to be triggered manually.

Avoiding Redundant "MH Location Information" Messages

In an environment where both forwarding mode and autonomous mode are utilized, a PFS might send unnecessary "MH Location Information" messages to SHs using forwarding mode. Because they are using forwarding mode, these SHs will ignore the "MH Location Information" messages. Packets from the SH to the MH will continue to be sent to the PFS, resulting in the generation of ineffective and unnecessary "MH Location Information" messages.

To avoid this, a PFS should keep a list of hosts it serves that are using forwarding mode. The PFS can then refrain from sending "MH Location Information" messages to any host on this list. Hosts can be added to this list when the first "MH Location Information" message cannot be delivered to the SH. The failure can be detected either by an ICMP message that indicating that the destination is unreachable [12] or when the SH fails to acknowledge the "MH Location Information" message.

Notification of Information

If "MH Location Information" messages can be sent to the network by directed broadcast, we will have the advantage of providing increased robustness in our location management mechanism. However, as RFC-1122 [7] mentions, the directed broadcast address may be unusable on some networks. Therefore we did not select to use it.

Other Issues

Adaptive Mode Selection

An MH that transmits a "Ping Autonomous Supporter" message may have to wait some time for a local PFS to reply. This delay is passed to

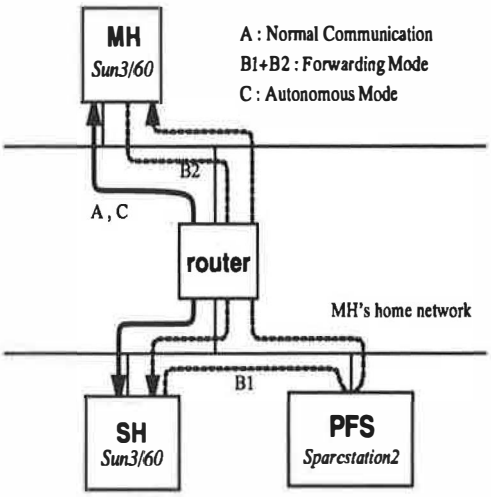


Figure 10: Case A

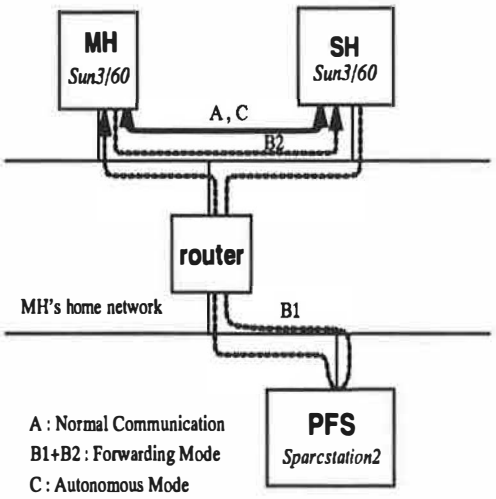


Figure 11: Case B

applications as additional latency introduced by MH migration. To avoid this problem, the MH may send the "MH Location Information" to the home PFS with no Autonomous flag set. After the MH finds a PFS which supports autonomous mode, it may send an "MH Location Information" message, this time with the Autonomous flag set.

Gateway Packet Filters

For security reasons, some gateways filter packets based on a certain field of the packets e.g., port number field [18]. Because an original packet is encapsulated in an IPTP packet in our approach, such gateways will fail to filter out packets that might otherwise be objectionable because the packet filters do not see within the IPTP packet. Similarly, a filter applied to IPTP will remove all encapsulated packets, regardless of how the local system administrator feels about them.

One way to solve this problem is to redesign the IPTP packet format. "Packet Transmission" messages could reflect the packet type in a newly defined IP option field rather than be indicated in the port number field.

Performance

We measured performance of packet forwarding on the following points:

1) End-to-end transmission rates both in the forwarding mode and in the autonomous mode.
2) Forwarding throughput of a PFS.

Measurement Conditions

The measurement was done between two Sun3/60 (SunOS 4.1.1) workstations. One SparcStation 2 (SunOS 4.1.2) workstation was used for a PFS. Normal TCP/IP transmission rate was also measured for comparison. We measured the transmission time of TCP/IP communication in two distinct conditions, which are illustrated as cases A and B in Figure 10 and 11, in normal TCP/IP, in forwarding mode and in autonomous mode. Case A is a case of an MH communicating with an SH in its home network after it migrated from its home network to another network, for example when it uses an NFS server in its home network after migration. Case B is a case of an MH communicating with an SH in the new network, for example when it uses a printer in the current network.

Results and Considerations

- **Transmission rates** Table 3 shows the result of the measurement on transmission rates. In case A, the transmission rates in forwarding mode and in autonomous mode were about 78% and 88% respectively compared with normal TCP/IP. Therefore we could estimate that overhead of packet encapsulation in an SH and packet decapsulation in an MH is included in 12%, and overhead of packet forwarding by a PFS and packet decapsulation in an MH is included in 22%. We think the data shows that there would not be a grave problem in practical usage.

	Normal	Forwarding	Autonomous
case A	1557 [Kbps]	1212 [Kbps]	1368 [Kbps]
case B	2617 [Kbps]	1059 [Kbps]	2256 [Kbps]

Table 3: Transmission rate

In case B, the transmission rate in forwarding mode is about 40% compared with normal TCP/IP. It is natural for the transmission rate in forwarding mode to be about half as much as that of normal TCP/IP, because a packet from an SH to an MH takes a very devious route, by which first it is sent to the MH's home network, and then is forwarded by the home PFS to the real destination. However the forwarding mode has a very significant point for applying to various existing

environments. In autonomous mode there is no overhead caused by the process of packet forwarding on a PFS; the delay time compared with normal TCP/IP is caused by the overhead due to enhancements on communication software in both hosts. Therefore the performance in autonomous mode is almost as good as in normal TCP/IP as the data shows.

- **Forwarding throughput** Figure 12 shows the results of the measurement of PFS's throughput. According to the definition by RFC 1242 [14], we define that the throughput is the maximum rate at which none of the incoming packets are dropped by a PFS. We performed six experiments varying packet length as a parameter. Ethernet packet lengths are 90, 256, 518, 1024, 1478, and 1518 bytes respectively. As you see in the Figure 12, the peak values of the throughput are 1291, 1079, 950, 562, 406, and 389 packets per second.

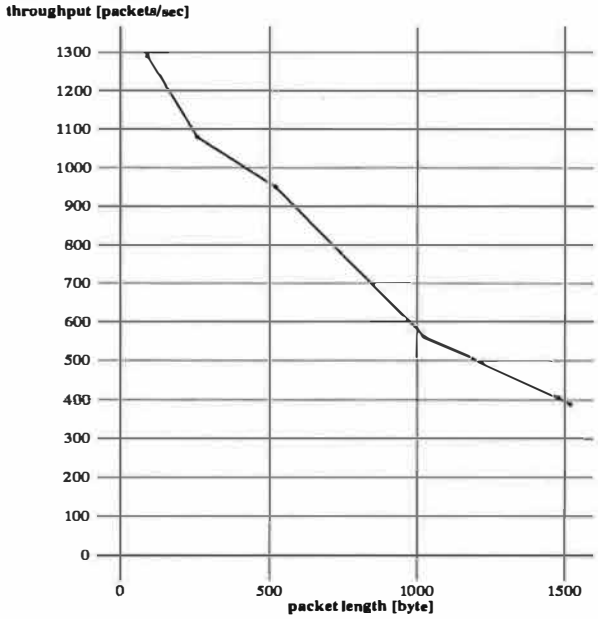


Figure 12: Throughput of a PFS

Recently the performance of Ethernet routers has been measured [15, 16]. For example the router "CISCO AGS+" has the following throughput; about 14500, 8300, 4500, 2300, 1200 and 800 packets per second for 64, 128, 256, 512, 1024 and 1518 packet length respectively. Compared with these results, the performance of the PFS in this experiment is lower, especially for packets with shorter length.

The cause of the difference is analyzed below. First of all, since a PFS has only one Ethernet interface, the forwarded packet stream is in the same Ethernet cable as the incoming packets stream. Hence even if it can handle all of the incoming packets, the throughput of the PFS can not exceed 5 Mbps. We expect that if the PFS is implemented on an IP routing station, the throughput will be better. As described in the section "Implementation", we put the IPTP module in user space in this PFS implementation, therefore we expect that if it is implemented in kernel space, the throughput will be significantly better.

- **Fragmentation effect** Finally, we estimate some fragmentation effect by forwarding. In this implementation, an IP packet with 1500 bytes size is fragmented when the PFS forwards the packet, because the PFS adds an IPTP header and an IP header to the IP packet for sending it to the proper destination. 1460 bytes size is the upper-bound not to be fragmented. The throughput on sending IP packets of 1500 bytes size is almost equal to that on sending 1460 byte packets (see Figure 12). That means that overhead by fragmentation on PFSs is negligible.

Considerations

Scalability

The focal point of scalability in our approach is a PFS. Scalability should be considered from the viewpoint of both workload placed on a PFS and the amount of MH location information to be maintained in a PFS. Sources of the workload placed on a PFS are:

1) Forwarding packets and processing IPTP messages for home MHs.
2) Forwarding packets and processing IPTP messages for visitor MHs.
3) Monitoring packets transmitted on the subnet.

Each workload for (1) and (2) depends on the number of home and visitor MHs respectively. More exactly, workload by a visitor MH remains beyond the MH's migration to another network. However it will be attenuated rapidly as the location information spreads over internet. Workload by (3) depends on packet traffic density in the subnet (this is because of the network monitoring workload in the promiscuous mode). The amount of location information to be maintained in a PFS also depends on the number of home and visitor MHs. However, the amount of the information for visitor MHs will be limited, because they should be maintained only as temporary caches. The workload to perform the forwarding service can be shared by PFSs distributed in the internet, and the amount of information for whose maintenance each PFS is responsible is limited only according to the number of its home MHs. Our method is scalable in that sense. Besides, the autonomous mode significantly lightens the workload of the PFS, because an SH communicates directly with an MH

except in the transient period. This fact enhances the capacity of a PFS.

Furthermore, it can be considered that multiple PFSs in a subnet share the service workload. Although that is no more than an idea so far, scalability can be improved by that.

Compatibility

As we have seen so far, an unmodified SH can communicate with an MH. An SH in forwarding mode completely communicates with an MH as if the MH were always at the home address. Also, the introduction of PFM puts no impact on existing routers basically, because it requires no protocol modification to the IP layer. There is the filtering issue of routers discussed in the section "Gateway Packet Filters". However, it does not seem to be a fatal problem.

A lot of traditional communication application programs use an IP address as an identifier of a host for access control. In PFM, not a temporary address but a home address of a client is shown to a server application program as an IP address of a mobile client, so such an access control scheme is still valid in the mobile computing environment.

Information Consistency

In forwarding mode, location information of an MH is needed only by its home PFS. An MH informs its home PFS of its location whenever it moves to another network. Location information consistency is maintained unless the communication path between the MH and the home PFS is unavailable.

In autonomous mode, the situation is more complicated than in the other mode. Location information of an MH is required by peer SHs and an visited PFS as well as by its home PFS. A home PFS always knows the newest temporary address of its home MHs, because it is informed whenever the MHs migrate. In autonomous mode, when a home PFS is informed of a new location from an MH, the PFS also informs the PFS which the MH previously visited of the new location of the MH. Notice that PFSs other than the last one previously visited and peer SHs are not notified this time. Hence the visited PFSs and the peer SHs can hold obsolete location information of an MH for a certain period. However, that will not cause any problem, because they are informed by lazy notification.

Those who are not informed of the newest address of an MH may send packets to the obsolete temporary address of the MH. If a PFS in the network to which the packets are destined caches the current location of the MH, it forwards them to the real address of the MH, and sends an "MH Location Information" message to the sender of the packets to notify it of the MH's newest temporary address. A visited PFS which forwards packets destined for the obsolete address may also have another obsolete (but newer than that of the sender) piece of location information. In this case, the process above takes place recursively. That means that the location information is propagated reversely along the path among which the MH migrates. If a PFS on this path has lost the location information of the MH by, for example, deletion of the cache entry, it can not tell the senders the current location information but informs them that the MH has gone elsewhere by an "MH Location Information" message whose "temporary address" field is NULL. When the sender receives the message, it invalidates its cache entry for the MH. If the sender is an SH, it routes the next packet for the MH to the home address of the MH, and consequently it is notified of the current location information of the MH by the home PFS. If the sender is a visited PFS, it simply invalidates the cache entry.

Thus, location information keeps enough consistency to route forwarding packets appropriately.

Robustness

Let us compare the robustness of the mobile environment of PFM with that of an ordinary (i.e., non-mobile) environment. Two hosts in separate networks can communicate with each other when both hosts and at least one route of the paths between the hosts are all available. The route consists of one or more gateways and links. The route can be dynamically selected avoiding an unavailable one. The forwarding mode could diminish the range of the route selection, because the route necessarily runs through the home PFS of the MH. Besides, it usually augments the length of the route. In such a sense, the availability of the communication between two hosts in the mobile environment can be relatively lower than that of the ordinary environment. In the autonomous mode, failure of the path that connects an MH, its home PFS and an SH could also be fatal if it happens while the MH location information is being distributed. However, these paths still can be made redundant by multiple routes using the ordinary dynamic IP routing mechanisms in both modes. Therefore, there is no essential difference in robustness between the two environments except that the home PFS can be the critical point of failure. It may be required that PFSs are highly available for this reason. We are investigating multiplexing of PFSs by which an arbitrary PFS can take over the load of a failed PFS.

Temporary failure of a home PFS or a path to it cannot be fatal. When the failure recovers, the home PFS will definitely obtain the current temporary address of an MH, because the MH sends

"MH Location Information" messages infinitely until one of them is replied to with an acknowledgement. It never loses track of MHs.

Failure of a visited PFS is not fatal at all. Communication with an MH can be recovered by the assistance with its home PFS.

Security

The most critical point of PFM regarding security is incorrect routing by the false location information. An intruder can confuse forwarding route or collect packets destined for an MH by distributing false "MH Location Information" messages. The protocol should be extended to include more strict authentication of the sender of the message in a manner that has integrity with common authentication mechanisms.

Related Work

A main characteristic of the PFM is that it provides a routing mechanism for MHs without affecting the existing routing mechanisms and a reliable MH location information management mechanism. We formerly proposed another migration transparent communication method [19] which mainly differs from this method in the addressing of mobile hosts. In that approach, a mobile host has multiple temporary addresses and no home address. However, the idea included some security problems. In order to solve them, we introduced the concept that a mobile host always has its home address, which appeared in [1].

OSI [9] and TCP/IP [10, 11] had not considered host mobility before. Recently, however, there is an approach which allows the current set of ISO/GOSIP standards to support the routing of Mobile End Systems(MES) [6]. This approach enables mobile networking while retaining the scaling benefit of hierarchical routing, although it requires a name service that updates the database in real time.

In Internet, Cerf formerly pointed out two problems regarding ambiguity of IP addresses [4]. He suggested that the partitioned network problem or the multi-homed internet host problem must be solved.

There is an approach which does not provide migration transparency, but provides quick propagation of the updated data of the name service [5]. If a partner host migrates after an application program establishes a name-address binding for the partner host, it is hard for the application program to adapt the name-address binding to the new situation.

Recently there are some approaches dealing with routing for a mobile host. Cohen discusses IP addressing and routing of a mobile host [13]. [13] provides a well-summarized overview of three schemes – the permanent IP-address scheme, the temporary IP-address scheme and the embedded network scheme.

We know of three approaches which focus on their concepts and discuss their mechanisms in detail. One approach is based on the introduction of a "virtual network layer protocol" named VIP(Virtual Internet Protocol) [1]. We derived the concept of a "home address" from [1]. In the VIP approach, some gateways are required to employ a VIP and some protocol conversion gateways are introduced for backward compatibility. We think it is one of the most important points whether a gateway is required to be modified or not. We choose unenhancement of gateways for backward compatibility.

Another approach is that of Columbia University [2]. In this approach, a group of MHs are required to belong to a same virtual sub-network, and MSR (Mobile Support Router) works as a gateway between the virtual sub-network and the other networks. It remains to augment optimum routing for mobile hosts.

There is also the IBM approach [3]. It uses loose source routing of IP option in order to get a route for an MH. MH location information is maintained by Mobile Routers(MR). We think the most advantageous point of the IBM approach is that it supplies optimal routing without enhancement of stationary hosts.

The last two approaches [2, 3] eliminate the "home/temporary address" concept at the expense of augmenting location information traffic exchanged over networks.

Summary

We have proposed the Packet Forwarding Method(PFM) which provides packet forwarding and host location tracking. We defined Internet Packet Transmission Protocol(IPTP) which is a mobile internet protocol based on the PFM, and experimentally constructed a mobile computing environment. Measurement shows that little communication overhead is incurred. Major advantages of the PFM are:

- **Compatibility with existing environments** No modifications to routers and no mandatory modifications to stationary hosts. (Enhancement of stationary hosts is optional.)
- **Efficient routing** Optimal forwarding route in autonomous mode.
- **Lower traffic load** Lazy propagation of migration notification.
- **Less complexity** Primary-copy management of replicated location information
- Furthermore, our methodology is characterized by the following feathers:

- Snoopy packet interception to allow mobile hosts and stationary hosts to be peers instead of requiring a distinguished gateway.
- Combination of a pair of addresses separating routing information from host identification to preserve addressing transparency.
- Maximal separation of location maintenance to minimize both the complexity of the clients and the complexity of maintaining up-to-date location information.

We see improving this method to support multiple PFSs for scalability and reliability and to guarantee security as the most major subject that remains to be investigated.

Acknowledgments

We would like to thank many people in Matsushita Information Technology Laboratory and Matsushita Electric Industrial Co., Ltd. for their help and useful comments. Especially we are very grateful to Brian Marsh and Masashige Mizuyama for their contribution in technical discussions. Without them this paper would not have been possible.

References

[1] F. Teraoka, K. Claffy, and M. Tokoro, "Design, Implementation, and Evaluation of Virtual Internet Protocol" In Proceedings of the 12th International Conference on Distributed Computing Systems, June 1992.

[2] J. Ioannidis, D. Duchamp, and G. Q. Maguire Jr., "IP-based Protocols for Mobile Internetworking" ACM SIGCOMM, September 1991.

[3] Y. Rekhter and C. Perkins, "Optimal routing for mobile hosts using IP's Loose Source Route option", Internet Draft, October 1992.

[4] V. Cerf, "Internet Addressing and Naming in a Tactical Environment", IEN-100, August 1979.

[5] S. Yanagishima and H. Sunahara, "Routing problem on the Internet with mobile node" The 16th UNIX symposium, December 1990.

[6] K. G. Carlberg, "A Routing Architecture that Supports Mobile End Systems", MILCOM'92.

[7] R. Braden, Editor "Requirements for Internet Hosts – Communication Layers", RFC 1122, October 1989.

[8] R. Droms, "Dynamic Host Configuration Protocol", Network Working Group, Draft RFC, November 1991.

[9] ISO. "Information processing systems – Open Systems Interconnection – Basic Reference Model", 1984, ISO7498.

[10] Postel, J., "Transmission Control Protocol", RFC 793, September 1981.

[11] Postel, J., "Internet Protocol", RFC 791, September 1981.

[12] Postel, J., "Internet Control Message Protocol – DARPA Internet Program Protocol Specification," RFC 792, September 1981.

[13] D. Cohen, J. Postel, and R. Rom, "IP Addressing and Routing in a Local Wireless Network", University of Southern California, July 1991.

[14] Bradner, S. O., "Benchmarking Terminology for Network Interconnection Devices", RFC 1242, July 1991.

[15] Bradner, S. O., "Application of Bridges and Routers: Network Design and Product Survey", Tutorial INTEROP'90, October 1990.

[16] Bradner, S. O., "LAB TEST / ETHERNET BRIDGES AND ROUTERS", Data Communications, McGraw-Hill, Inc., 1992.

[17] SunOS Release 4.1 manual NIT(4P), December 1987

[18] CISCO Systems Inc., "Gateway System Manual", November 1990.

[19] H. Wada, T. Yozawa, T. Ohnishi, and Y. Tanaka, "Migration Transparent Communication Method Based on Packets Forwarding", The Technical Report of IEICE, SSE92-67, IN92-58, September 1992.

Author Information

Hiromi Wada received her B.S. in Chemistry at Osaka City University in 1983. She then joined Matsushita Electric Industrial Co.,Ltd. She is interested in distributed computing environments, especially mobile computing. Her e-mail address is: hwada@isl.mei.co.jp.

Takashi Yozawa received his B.S. and M.S. degrees in Mathematics at Osaka University in 1989 and 1991 respectively. He then joined Matsushita Electric Industrial Co.,Ltd. Currently he is engaged in research regarding communication in mobile environments. His e-mail address: yozawa@isl.mei.co.jp.

Tatsuya Ohnishi received his B.S. and M.S. degrees in Information Science at Kyoto University in 1986 and 1988 respectively. He joined Matsushita Electric Industrial Co,. Ltd. His main area of interest is networking, especially mobile computing. His e-mail address is: ohnishi@isl.mei.co.jp.

Yasunori Tanaka received his B.S. and M.S. degrees in Computer Science at Osaka University in 1982 and 1984 respectively. He joined Matsushita Electric Industrial Co,. Ltd. He is interested in personal communication, distributed

management environments and mobile computing environments. His e-mail address is: tanaka@isl.mei.co.jp.

All are researchers in Information Systems Research Laboratory at Matsushita Electric Industrial Co., Ltd. Our address is: Information Systems Research Laboratory; Matsushita Electric Industrial Co., Ltd.; 1006 Kadoma, Kadoma-shi, Osaka 571, JAPAN.

The Compression Cache: Using On-line Compression to Extend Physical Memory

Fred Douglis – Matsushita Information Technology Laboratory

ABSTRACT

This paper describes a method for trading off computation for disk or network I/O by using less expensive on-line compression. By using some memory to store data in compressed format, it may be possible to fit the working set of one or more large applications in relatively small memory. For working sets that are too large to fit in memory even when compressed, compression still provides a benefit by reducing bandwidth and space requirements.

Overall, the effectiveness of this *compression cache* depends on application behavior and the relative costs of compression and I/O. Measurements using Sprite on a DECstation 5000/200 workstation with a local disk indicate that some memory-intensive applications running with a compression cache can run two to three times faster than on an unmodified system. Better speedups would be expected in a system with a greater disparity between the speed of its processor and the bandwidth to its backing store.

1 Introduction

Over the past decade, the processing power and physical memory size of typical computers have increased dramatically. Even as workstation memory sizes are increasing, however, a new technology trend is pushing toward small memories: mobile computers that are smaller than their desktop counterparts and are typically configured with significantly less memory. Application designers are sometimes forced to squeeze their applications to fit into available memory, and may not succeed. Therefore, in a general-purpose mobile computer, as with many computers, paging is needed to enable a wider range of applications to run – as long as it can be performed efficiently.

The difficulty in paging on mobile computers arises from similar technology trends. While workstations are normally connected to relatively fast local-area networks and moderately fast disks, mobile computers may communicate over slower wireless networks and run either diskless or with small, slower local disks. At the same time, however, the processors on mobile computers are steadily improving in speed, and the disparity between processor speed and I/O speed is at least as great for mobile computers as for workstations. This disparity suggests a new technique for managing memory, which exploits *compression* to reduce I/O.

Compression is already widely used to reduce demand for secondary storage and networks. I suggest that it is now feasible to use compression to reduce the demand for memory as well. The basic idea is to take some memory that would normally be used directly by an application, and use it instead to hold a larger number of pages in compressed format. I call the area used for compressed data a *compression cache.* If the pages touched by a process could not normally fit in memory, but could fit into memory when some were stored in the compression cache, then the processor would never have to write a page to backing store (onto a local disk or over a network to another computer). Even if pages must be written to backing store, compressing them beforehand reduces the amount of data transferred.

The potential benefits of the compression cache depend on the relationship between the speed of compression and the I/O bandwidth of the system, as well as the compression ratio (anywhere from barely over 1:1 to about 4:1 in the experiments reported below). If the cost of compressing and copying a page were negligible, and pages compressed well, the compression cache could be used to give a computer the appearance of having additional physical memory. In practice, compressing and copying have costs associated with them, and the benefit of reducing traffic to the backing store is offset by the overhead of the compression cache. Overhead comes not only from the compression itself but from the additional page faults an application will experience when some memory is used for compressed pages (as well as the data structures used to support compressed pages). Note that these page faults need not require I/O to be expensive – merely decompressing and compressing many pages that would otherwise be immediately accessible to the application can degrade performance. Furthermore, as mentioned above, not all applications compress well: for poorly suited applications, the effort to compress memory will be wasted and degrade rather than improve performance. Thus, depending on the application and the hardware environment, the benefits of reduced I/O may outweigh the costs of compression and additional faults, or vice-versa.

Configuring the compression cache to improve performance in the first case while staying out of the way in the second case is an interesting, and difficult, problem.

The remainder of this paper is organized as follows. Section 2 discusses related work involving paging or compression. Section 3 elaborates on the tradeoffs involved with compressed paging. Section 4 describes the design of the compression cache, based on these tradeoffs. Section 5 evaluates the performance of the compression cache for some sample applications. Finally, Section 6 concludes the paper.

2 Related Work

This section discusses other projects and products with goals similar to the compression cache. They fall into two general categories: file systems and virtual memory.

File Systems

A number of systems have replaced *ad hoc* techniques for manual compression with a mechanism for automatically compressing some or all files. Cate and Gross used compressed files as a level in a hierarchy, with recently-accessed files being in uncompressed format and less-recently-used ones compressed [5]. Since frequently-used files were never compressed, and compression was performed in the background, the overall impact on interactive performance (delays due to decompression) was minimal: less than 50 seconds per user per day. At the same time, disk space requirements were roughly halved.

Burrows, et al. integrated compression with Sprite LFS [12], also primarily to reduce disk space requirements [4]. They argued that LFS is a better vehicle for compressing files than traditional file systems, since files are not overwritten in place and a change to one block within a file would not cause changes to compressed data later in the file. Multiple file blocks may be compressed as a unit, providing better compression than if each block were compressed separately using a dynamic compression algorithm such as LZRW1 [16]. Burrows, et al. found that on-line compression halved disk space requirements, as in Cate and Gross's system, without the delays that could be incurred by decompressing a large file as a single unit. The system had an acceptable performance degradation when compression was performed in software, and was well-suited to hardware compression.

In addition, there is a family of products for personal computers that do both on-line and off-line compression for the purpose of reducing disk space usage. A discussion of these products is available elsewhere [4], so it is omitted here.

Virtual Memory

The focus of the above systems has been disk space rather than performance. Other projects have considered ways not only to reduce disk space demands, but also to improve performance, particularly in the area of virtual memory.

Taunton described a mechanism for compressing binary executables on Acorn personal computers, reducing disk space requirements and improving file system bandwidth [14]. Because compression of the executables was performed off-line, an especially effective (but slower) compression algorithm was available. Bandwidth improved because the cost of decompression was offset by the reduction in data transferred from the disk. As a result, the performance of program invocation improved.

Atkinson, et al. at Xerox PARC, considered the use of compression in order to reduce the cost of paging over wireless links [2]. Such paging might be needed in an environment with mobile computing devices that are too small to have local disks, such as the "tabs" advocated by Weiser [15]. The PARC researchers concentrated on read-only data, such as executables, because of the space and time overhead of performing on-line compression. Executables would be stored and transmitted in compressed format, and cached on a mobile computer in compressed format to increase the number of such executables that could be cached. As in Taunton's system, because compression would be performed off-line, an asymmetric compression algorithm could be used that would give very good compression ratios (with a correspondingly high overhead) while decompressing quickly. These researchers did consider on-line compression as well, resulting in a suggestion (reported by Appel and Li [1]) that pages be compressed and retained in memory. This idea, which they did not pursue extensively, is the primary theme of this paper.

3 Design Considerations

Intuitively, the idea of trading processing (compression) for I/O is appealing: by and large, processors are improving in performance more quickly than I/O devices, especially disks.[1] If one can compress some pages so that they occupy little enough memory to permit all of a process's address space to reside in memory, it might be possible to avoid I/O to the backing store completely; the

[1] Paging over a network rather than to a local disk is another issue. In some environments, it is more efficient to page over a 10-Mbps Ethernet to memory on a file server than to page to a local disk [9]. Some local-area networks, such as ATM networks (e.g., Autonet [13]), provide bandwidth that is at least an order of magnitude greater than an Ethernet. However, for mobile computers on wireless networks, one can expect the disparity between processing and I/O to remain for some time.

process would execute correspondingly faster. Note that this technique is fundamentally different from writing a dirty page into a file system that does compression, or a disk that does compression at the driver level, because compressed pages never have to go to backing store at all. Instead, compressed pages form an intermediate level in the storage hierarchy, between uncompressed pages and the backing store.

Keeping compressed pages in memory does not obviate the need for a backing store, however. It is possible for the collective address space of all running processes not to fit in memory even after compression. And even if they fit, it might be desirable to move some "old" pages to backing store in order to have more memory available for actively-used pages. In either case, pages could be transferred to backing store in compressed format, reducing the demand for bandwidth. This technique would be similar to paging into a file system or disk that does its own compression. The differences are:

Reduced I/O With the compression cache, some pages might be faulted upon before being written to backing store. Those pages would no longer need to be written.

Variable memory allocation By making compressed pages an explicit part of the memory hierarchy, the system can dynamically vary the amount of memory used for uncompressed pages, compressed pages, and file blocks. This is necessary to avoid impacting applications that do not need to compress pages, as discussed below in Section "Variable Memory Allocation".

Complexity and space overhead These are better when compression is performed at the level of the backing store, rather than the VM system. Assuming a one-to-one mapping between VM pages and file blocks, transferring pages that are already compressed requires the VM system to cluster multiple compressed pages into a smaller number of file blocks. Also, the VM system must manage free space on the backing store at a granularity finer than individual file blocks. Considering that the file system may use compression regardless of its use for virtual memory, the extra overhead to manage the backing store may be wasted effort and wasted memory. This issue is discussed further below.

Regardless of whether compression is performed explicitly by the VM system or implicitly when pages are transferred to backing store, the effectiveness of compressing VM pages depends on several factors:

Compression speed Compressing a page, and later decompressing it, must be significantly faster than transferring it to or from backing store. Otherwise, one might as well do traditional paging without compression.

Compression ratio On average, pages must compress to significantly less than their original size. Obviously, compressing a 4-Kbyte page to 3500 bytes is far less useful than compressing it to a few hundred bytes.

Page access patterns If pages are compressed and retained in memory, then less memory is available for uncompressed pages. An application will likely take additional page faults, accessing pages that would be resident and uncompressed in a standard system but are instead stored in compressed format. Given this effect, it is important that the compression cache not degrade performance: if the collective working set of active processes fits into physical memory without the need to compress pages, the compression cache should stay out of the way. This implies that its size should vary dynamically over time as the demand for memory changes.

Memory overhead Keeping pages in both uncompressed and compressed format has memory overhead associated with it (keeping track of the state of each page, as well as where pages are stored on disk). Taking this memory away from applications also results in additional page faults.

Compression implementations The compression cache should allow for both software- and hardware-based compression. Ideally, it should allow different compression algorithms to be used for different types of data, in order to get the best compression rates and/or throughput.

As one might expect, there is an inverse relationship between compression speed and compression ratio: the faster a page is compressed, the less compression is required for compression to improve performance. Figure 1(a) graphs the speed of paging to and from backing store in compressed format, as a function of compression bandwidth (relative to the bandwidth of the backing store) and compression ratio. Figure 1(b) shows the speedup of mean memory reference time as a function of these two variables, when pages are retained in memory, for an application that sequentially accesses twice as many pages as fit in memory, reading and writing one word per page. In this case, if pages are compressed to no larger than half their original size, on average, the speedup due to compression is linear in the speed of compression. Of course, if pages do not compress well, then compression must be much faster than I/O or overall performance will be worse than without compression. In some systems it is also possible for an application to issue an "advisory" to the operating system to indicate that least-recently-used (LRU) page replacement will

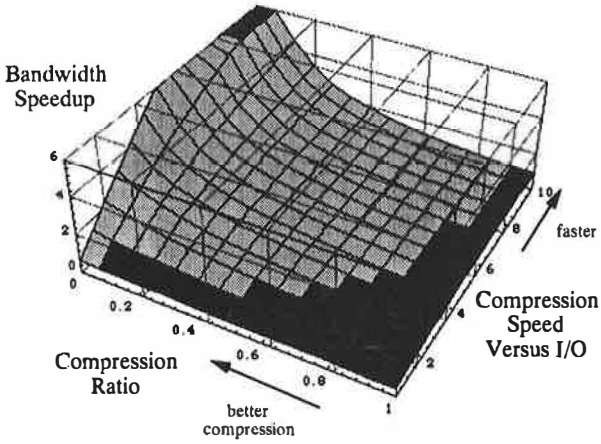


Figure 1a: Transferring compressed pages to backing store

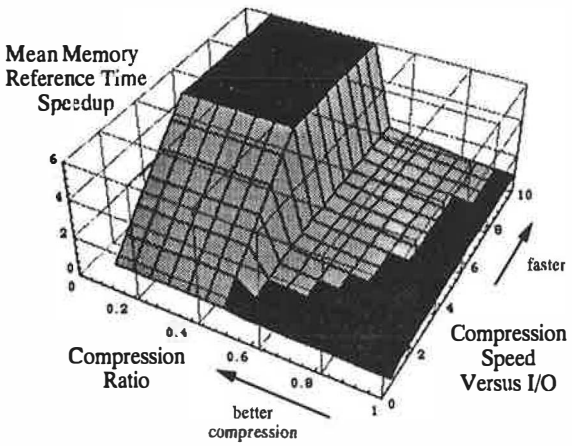


Figure 1b: Keeping compressed pages in memory

Figure 1: Performance of compressing pages, modeled analytically. Speedups are shown as a function of the compression ratio (fraction of bytes left after compression) and the speed of compression relative to I/O. Decompression is assumed to be twice as fast as compression, as is roughly the case for algorithms such as LZRW1 [16]. There are three regions of speedup: the dark black areas at the top left show speedups that go off the top of the scale (6-fold improvement); the light areas show speedups of 1-6 relative to no compression, and the darker areas to the right show data points at which a slowdown would result.

behave poorly; in this example, half the pages could effectively be pinned in memory with faults occurring only on the other half. With fast compression, however, even reducing I/O by a factor of two will be inferior to keeping all pages compressed in memory.

The sharp leap in speedup when all pages fit in memory, as in Figure 1(b), demonstrates the potential difference between the compression cache and a system that compresses pages en route to the backing store. In practice, this improvement is not fully realized, because access patterns are not so pathological. The performance of sample applications is given below, after the description of a specific implementation of a compression cache for the Sprite operating system [11].

4 Design

This section describes the design and implementation of a compression cache in Sprite. Sprite is largely compatible with 4.3 BSD UNIX, but its virtual memory system has an interesting difference from most versions of UNIX: physical memory is traded dynamically between VM for application processes and the file system's buffer cache [9]. Since the compression cache must vary in size dynamically as well, Sprite provides a good framework for prototyping the compression cache. The idea of the compression cache should extend naturally to UNIX, Mach, or other systems; in fact, Mach's external pager interface [7] should be an excellent foundation for future work in this area.

The target environment for this research consists of mobile computers with limited memory and network bandwidth, and with small local disks or no disks at all. Because of the limited availability of Sprite, the compression cache has been prototyped in a workstation environment, running on DECstation 5000/200 workstations, paging to a local RZ57 disk. The Sprite kernel is configurable at boot-time to allow the system to use a variable amount of physical memory, so a 32-Mbyte machine can behave as though it has as little as 12 Mbytes. About 6 Mbytes are used by the kernel for code, page tables, and some forms of tracing that cannot currently be disabled.

Overview

The compression cache forms a new level in the memory management hierarchy. A general description of the technique is as follows.

- LRU pages are compressed to make room for new pages. The compressed pages are retained in memory for a period of time, in the expectation that they will be accessed again soon.
- If not all pages fit in memory, even with some compressed, the LRU compressed pages are written to backing store.
- To service a page fault for a page that is not already uncompressed and resident in memory, the VM system checks to see whether the page is compressed in memory or on the backing store. If it is on backing store, it is first brought into memory and stored in the compression cache, then it is decompressed and made accessible to the faulting process. The compressed copy in memory can be freed at any time, since there is already a copy on backing store.

Specific issues arise in a number of areas. First, the VM system must be able to vary the amount of physical memory allocated to the compression cache, taking into account the demand for uncompressed pages and for the file system buffer cache. Second, the interface between the compression cache and the backing store is complicated by the notion of variable-sized pages. Finally, the overhead of managing the compression cache should not adversely affect performance. The following subsections discuss these issues.

Variable Memory Allocation

Initially, the compression cache was implemented as a fixed-size region of physical memory. This was done partly for simplicity and partly because the need to vary its size was not yet apparent. In this version, the compression cache consisted of a number of pages, each divided into N fragments (In my experiments, N was defined to be 8, meaning blocks of 512 bytes with a pagesize of 4 Kbytes). When a page was compressed, the system allocated enough fragments to hold the compressed data. The fragments did not need to be allocated contiguously; instead, the compression was performed into a contiguous buffer and the compressed data was then scattered into multiple fragments. To satisfy a page fault, the fragments for a page were copied into a contiguous buffer and then decompressed.

The fixed-size implementation was simple, since unused fragments could be linked together on a list, and fragmentation could be kept to a minimum. But this implementation was suitable only for applications that paged heavily even without the compression cache, and which fit into the compression cache without excessive traffic to the backing store. For example, on a machine with 8 Mbytes of memory available to user processes, setting aside 4 Mbytes for compressed pages would cause a 6-Mbyte process to page, ruining its performance. On the other hand, even after compression a 12-Mbyte process probably would not fit into the 4 Mbytes available. In the first case, either no compression cache or a cache of less than 2 Mbytes would be better, but in the second case using 6-8 Mbytes for compressed pages might eliminate all traffic to the backing store.

The compression cache was therefore redesigned to vary its memory usage over time. At first I considered an extension of the previous design, with fixed-size page fragments, but there is a problem with this approach. To reclaim a whole physical page from the compression cache, to use for an uncompressed VM page or a file block, each fragment within the page must be copied elsewhere or written to backing store. Since a page with N fragments could contain a small piece of each of N different pages, either the physical page would be transferred directly to backing store (resulting in multiple I/Os to read all the fragments for a particular page upon a page fault) or several different pages would have to be transferred to backing store in order to free one physical page. In addition, the overhead of doing "scatter/gather" between the contiguous compression buffer and the page fragments is unnecessary.

Instead, memory for the compression cache is now treated as a variable-sized circular buffer. Physical pages are mapped into the kernel's virtual address space, one after another, eventually wrapping around to the start of the range of addresses for the compression cache. There is a notion of the *oldest* physical page – the one added to the cache the longest time ago – and new pages, which have been added most recently and may not contain data yet. Physical pages are added to one end of the queue and normally removed from the other end. (They may be removed from the middle if no clean pages are availabled at the oldest end.) When VM pages are compressed, they are compressed directly into the first unused region within the compression cache, following the last page that had been added to the cache. Before each page there is a small header that describes the page, the size it compressed to, whether it contains dirty data, a link to the next page in the cache, and other information.

Figure 2 shows a simplified view of the compression cache. Pages are in one of four states:

- **clean**: A page that has had all modified compressed pages within it written to backing store. A page can also be clean if it contains only compressed pages that have been brought in from backing store to satisfy page faults.
- **dirty**: A page with modified data that are not on backing store.
- **free**: A slot in the compression cache that does not have a physical page associated with it.
- **new**: A slot with physical memory allocated to it, but which does not yet contain data. *New* pages can only exist at the tail of the queue.

Pages may also be reclaimed dynamically from the compression cache. The oldest page in the cache with unmodified data is unmapped and returned to the kernel's pool of free physical pages. A kernel thread writes out the oldest dirty data in the compression cache in an attempt to keep a pool of physical pages clean and ready for reclamation. The rate at which pages are cleaned is a function of the number of completely free pages in the system, the number of clean pages that are already reclaimable, and the size of the compression cache.

The method for choosing when to grow or shrink the compression cache is similar to the algorithm in Sprite for trading memory between the file system and VM system. Sprite compares the age of the least-recently-used file block to the age of the LRU VM page, and reclaims the older of the two,

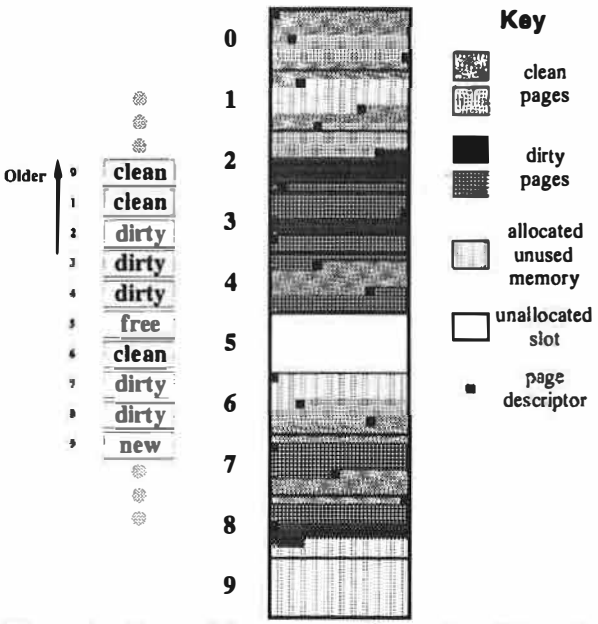


Figure 2: State of the compression cache. Physical pages may be in any of several states. An separate array of page descriptors stores the mapping of slots in the compression cache to physical pages and keeps track of the state of each page. Two stipple patterns represent distinct VM pages within the compression cache. Each user page has a small descriptor just before it indicating its state. Lighter pages are clean, while darker ones contain modified data. White areas contain no current data.

modulo an adjustment to favor retaining VM pages longer. This "penalty" to the file system helps improve interactive performance, by preventing a large file from flushing a process's address space completely out of memory [10].

With the compression cache adding a third collection of pages, and a third consumer of memory, the tradeoffs are more complicated. In the current implementation, allocation of each of the three types of memory (file system cache blocks, uncompressed VM pages, and compressed pages) requires a comparison of the ages of the oldest pages for all three types. The system biases the ages to favor compressed pages over uncompressed pages and both of these over file cache blocks. The more the system favors compressed pages, the larger the compression cache will tend to grow in periods of heavy paging; with a very low bias (or a bias in favor of uncompressed pages), the compression cache degenerates into a buffer for compressing and decompressing pages between memory and the backing store.

Interestingly, although a single penalty between VM and the file system works well across a wide range of applications, the optimal penalty for the compression cache is application-dependent. An application that exhibits a great deal of locality should have as many pages uncompressed at once as possible; thus the compression cache should serve just to buffer I/O to and from the backing store, but would not be expected to eliminate the I/O completely. Also, a large application that exhibits so little locality that its faults are rarely satisfied within the compression cache will not benefit from a large cache. Only an application with characteristics that cause it to "hit" in the compression cache will benefit from a large cache. Examples of such an application appear in the next section.

Interface to the Backing Store

In an unmodified Sprite system, the size of a VM page is an integral multiple of a 4-Kbyte file system block. Since on the DECstations (where the compression cache was prototyped) a VM page is 4 Kbytes, this discussion assumes a one-to-one mapping between file blocks and VM pages. When a page is written to backing store, it is written to a "swap file" corresponding to the segment containing the page, at an offset corresponding to the location of the page within the segment. This fixed mapping of pages to file blocks makes it trivial to locate a page on the backing store.

There are a number of ways to transfer variable-sized compressed pages to and from backing store, none of which is especially appealing. Ideally, the system would keep each compressed page in the same location in its swap file as without the compression cache, but transfer just the amount of data occupied by the compressed page. Unfortunately, with the exception of the last block in a file, the file system enforces transfers in multiples of a whole file system block. If part of a block is written then the file system reads the old contents and overwrites the part just written before writing the whole block back to disk. In other words, if a page were compressed from 4 Kbytes to 2 Kbytes, a 2-Kbyte write would result in a 4-Kbyte read and a 4-Kbyte write rather than only the expected 2-Kbyte write. Furthermore, a request to read 2 Kbytes within a 4-Kbyte block would result in the file system reading all 4 Kbytes and then copying just the part requested into the requesting process's buffer.

Without changing the internal structure of the Sprite file system, or writing every page into its own file (with significant overhead of its own), there is no way to avoid reading a minimum of 4 Kbytes to satisfy a page fault. This has the unfortunate effect of reducing the usefulness of the compression cache for applications that read a large number of pages in an unpredictable order: each page fault will require both a full 4-Kbyte read and a decompression. There are, however, possible solutions to the extra overhead for partial *writes* described above:

- A partial solution would be to issue an operation to write an entire block, thus writing 4 Kbytes without the need to read data from

disk beforehand. However, this would not benefit from having already performed compression. In an environment in which few pages are written to backing store this would be unimportant, but not all applications fit in memory even when compressed.

- Another possibility would be to modify the file system to overwrite part of a file system block on disk without reading the remainder of the block. In this case disk bandwidth would improve, but it would still suffer from having independent small I/Os rather than a small number of large I/Os. (Note that it might be possible to page into Sprite LFS [12], which provides much higher bandwidth by coalescing many small writes into a single larger transfer, but LFS suffers from the same restriction of 4-Kbyte transfers.)
- The solution I implemented attempts to transfer exactly the amount of data a page occupies when compressed, by merging several compressed pages into a smaller number of file blocks. This reduces fragmentation, with a corresponding reduction in bandwidth needs and disk space usage.

Merging compressed pages together has its own problems, however. First, this scheme loses the one-to-one mapping between offsets in a swap file and pages within a segment. Instead, it is necessary to store the location of each page explicitly. Second, when a page is written out to backing store, faulted back into memory, modified, and written out again sometime later, it may not be written to the same location. (If it were, the same problem of writing partial file blocks would occur.) Thus it becomes necessary to perform garbage-collection on the backing store to keep track of which blocks contain the most recent copy of page and which blocks contain obsolete data. If compressed pages can be written to arbitrary locations within a block, keeping track of the location and size of each page becomes a bookkeeping nightmare. Third, if pages are allowed to span two file blocks, it becomes necessary to read in both blocks to satisfy a page fault. Thus a 4-Kbyte `read` becomes an 8-Kbyte one. If page accesses exhibit sufficient locality that retrieving 8-Kbytes of compressed pages satisfies additional page faults without more I/O, spanning pages is not disadvantageous in the long run, but without this locality the system will pay a performance penalty.

The version of the compression cache I have implemented in Sprite pads each compressed page to a uniform fragment size (currently 1 Kbyte), and writes a set of fragments, spanning several file blocks, in a single operation. Currently 32 Kbytes of compressed pages are written at once. The system is parameterized to determine whether pages are allowed to span file block boundaries: if they cannot, then fragmentation increases and the effective bandwidth for writes to the backing store correspondingly decreases.

Overhead

The compression cache adds some overhead in terms of memory usage. The kernel sets aside a static buffer that is used for the LZRW1 algorithm's hash table [16]. This hash table can be relatively small, or it can be relatively large (e.g., on the order of 1 Mbyte), which improves compression at the cost of memory. In the system measured for this paper, the hash table is 16 Kbytes. In addition, the difference in code sizes between the unmodified system and the system with the compression cache is an additional 22 Kbytes.

When a segment is created or enlarged, its page table is essentially extended by 8 bytes per 4-Kbyte page, which is used by the compression cache. While this is only 0.2% overhead for pages that are resident in memory, this information is resident even when pages are not: for non-resident pages, an unmodified system stores just 4 bytes per page, rather than 12 for the compression cache. As an example, if the collective virtual memory of all running processes is 60 Mbytes, with 4-Kbyte pages, the per-page overhead for the compression cache would total 120 Kbytes.

There is also overhead for the space managed by the compression cache itself. The kernel uses 8 bytes per page in the range of addresses the compression cache might occupy (as shown in Figure 2). This overhead is determined at boot time based on the maximum possible size of the cache. The kernel also allocates a 24-byte header within each physical page frame that is mapped into the cache (0.6% overhead), and a 36-byte header for each virtual page that has been compressed and placed in the cache. These overheads occur only when the compression cache has data in it, and are offset by the savings in memory usage due to compression.

5 Performance

As Figure 1 showed, the improvement due to compression depends on the speed of compression, the amount of compression obtained, and the number of transfers to or from backing store that can be completely eliminated. I consider both the maximum possible performance improvement and the performance of some applications.

Maximum Possible Improvement

It is possible to estimate the maximum possible improvement for a particular configuration and compression algorithm by running an program that is contrived to thrash the VM system. *Thrasher* cycles linearly through a working set, reading (and optionally writing) one word of memory on each page each time through the working set. The system uses an LRU algorithm for page replacement, so

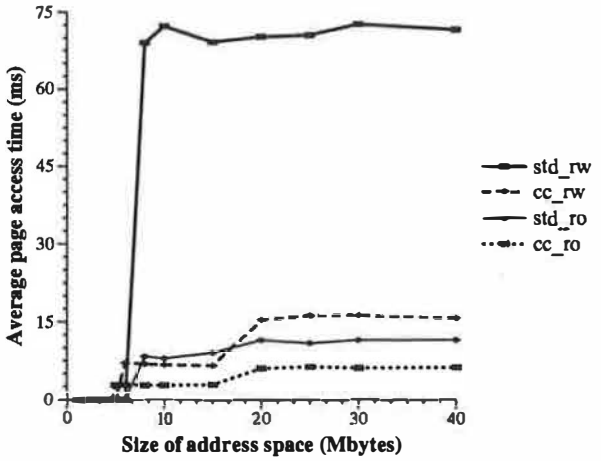


Figure 3a: Average page access cost

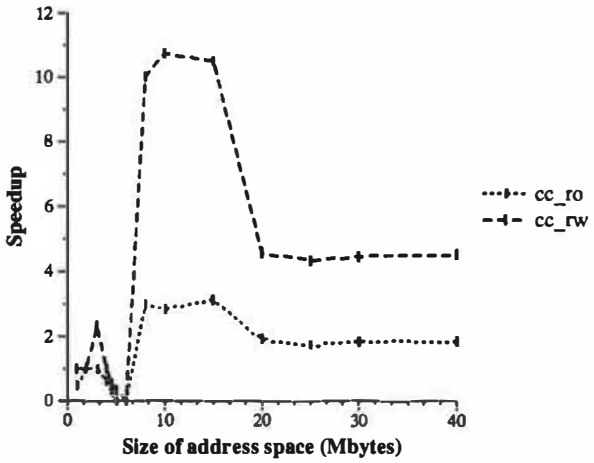


Figure 3b: Speedup relative to original system

Figure 3: Compression Cache Performance Under Thrashing. With an unmodified system, a large number of pages fit in memory without measurable page-fault overhead, but once the system starts thrashing it pays for one or more disk operations per page access. With the compression cache, pages compress roughly 4:1. Compression reduces the average access time considerably, especially when compressed pages fit in memory without the need for disk I/O (up to a total address space of about 15MB). Larger address spaces, from 20 Mbytes upward, resulted in disk I/O, but with fewer transfers and fewer seeks than the unmodified system. Measurements were taken on a DECstation 5000/200 with approximately 6 Mbytes available for user processes, paging to a local RZ57 disk, with a page size of 4 Kbytes. Compression was performed using Williams's LZRW1 algorithm. The labels are explained in the text.

if *thrasher*'s working set does not fit in memory, then it takes a page fault on each page access. If *thrasher* is modifying pages as it accesses them, the system must write a page each time to make room for the page being faulted on. The unmodified Sprite system, which uses regular files as the backing store, would perform two disk seeks for each fault, one to write a page out and another to retrieve the page faulted upon. If *thrasher* is only reading each page, then no seek is necessary if the pages are close to each other in the swap file (equivalently, close to each other in the address space).

With the compression cache, *thrasher* would still fault on each page, but each fault could be satisfied by a decompression (and a compression, if pages are being modified) rather than one or two disk I/Os. Since taking some memory for compressed pages does not increase the fault rate – *thrasher* was faulting on every new page anyway – the ratio between compression speed and I/O speed determines the speedup. If the working set does not fit in memory when compressed, then each fault may require a read from the backing store, as well as possibly a write to make room for it. By clustering compressed pages together, however, transfers are effectively smaller, and multiple pages can be obtained with a single read from the backing store. This can reduce the average number of seeks per page fault considerably.

Note that Sprite LFS could alleviate the problem of seeks between pageouts by grouping multiple pages into a single segment. However, it is not clear that paging into LFS would be desirable under heavy paging load. LFS requires significant memory for buffers, and for LFS to clean segments containing swap files, it must copy more "live" blocks than for other types of data [12].

Figure 3 shows access time as a function of working set size, on a machine configured to use no more than 12 Mbytes (of which about 6 Mbytes are available to user processes). There are four lines:

- **std_rw**: An unmodified Sprite system, sequentially reading and writing each page.
- **cc_rw**: A Sprite system with a compression cache, sequentially reading and writing each page.
- **std_ro**: An unmodified Sprite system, sequentially accessing each page without modification;
- **cc_ro**: A Sprite system with a compression cache, sequentially accessing each page without modification;

Figure 3(a) gives the average time to access each page, and Figure 3(b) gives the speedup of the compression cache relative to the original system.

Application Speedup

The speedup of *thrasher* gives an upper bound on the performance improvement real applications might experience through the compression cache. This is because *thrasher* almost always takes the same number of page faults even when some memory is set aside for compressed pages, and the memory *thrasher* accesses compresses extremely well. Real applications may not compress as well,

and they often exhibit a degree of locality that significantly increases their page fault rate as they are allocated less memory. This section reports the performance of some sample applications. A summary of these results is in Table 1.

One example of an application that exhibits a substantial improvement from the compression cache is a program that computes the sequence of modifications to change one file into another. *Compare* could be useful for transferring "diffs" rather than entire files, when the changes between two versions of a file are minimal. Lopresti implemented file differencing using a dynamic programming algorithm (refer to Lipton and Lopresti [8] for a description). The application uses a two-dimensional array, of which only a wide stripe along the diagonal is accessed. It works its way through the array in one direction, and then reverses direction and goes linearly back to the beginning. Elements along the diagonal are based on a recurrence relation that causes frequent repetitions in values, which in turn suggests that the data in the array are extremely compressible. With LZRW1 the pages compress about 3:1, and with other compression algorithms, the pages should compress even better.

Another example of an application that benefits from the compression cache is Dubnicki's cache simulator, which is both CPU-intensive and memory-intensive [6]. In a sample run, *isca* experienced a 50% improvement in execution time, and pages that were compressed during its execution averaged a 3:1 compression ratio as well.

Despite these two examples of applications with good compression ratios and consequently good performance using the compression cache, in general applications do not necessarily compress especially well, and their performance suffers accordingly. I considered an application that performs `quicksort` on a file containing approximately 12 Mbytes of text (numerous copies of each word in `/usr/dict/words`). If the text were completely unsorted to begin with (*sort_random*), so there was minimal repetition of strings within an individual 4-Kbyte page, the *sort* program ran significantly more slowly on the compression cache than the unmodified system – primarily because about 98% of the pages compressed fairly poorly – less than 4:3 (the threshold for keeping them in compressed format). Thus, the time to compress these pages was wasted effort. For the sake of comparison, *sort* 's heap compressed much better if the input file contained frequent repetitions of words – for example, if the input file were only a minor permutation of the sorted copy of the file, with substrings (or complete words) often repeated within a page of memory (*sort_partial*). In this case the compression ratio was about 3:1 and the application ran 23% faster than on the unmodified system (rather than 10% slower).

Finally, one might expect that a main-memory database would benefit from the compression cache if it fits in memory when compressed but not otherwise. Some accesses would be to data that tends to remain uncompressed ("warm" data), while others would be to less frequently used ("cold") data, which would stay mostly compressed. Each access to compressed data would incur the overhead of decompression (and subsequent compression if the page is modified), but not a disk I/O. However, the hit rate on uncompressed data would be lower than the hit rate in a system without the compression cache, because some memory would be used for compressed pages instead of regular virtual memory. The poorer the compression ratio, the greater the penalty.

Indeed, one such database, the "index engine" for the Gold Mailer [3], compresses slightly worse than 2:1; it runs more slowly under the compression cache than on an unmodified system. This is partly due to the poor compression and partly due to the high fraction of nonsequential page accesses it encounters, each of which requires a full 4-Kbyte

Application	Time (std)	Time (CC)	Speedup	Compression Ratio (%)	Uncompressible pages (%)
compare	16:14	6:04	2.68	31	0.1
isca	43:15	27:00	1.60	32	1.7
sort_partial	13:32	10:24	1.30	30	49
sort_random	26:17	28:51	0.91	37	98
gold_create	14:03	15:38	0.90	59	42
gold_cold	45:30	56:36	0.80	60	10
gold_warm	35:56	49:00	0.73	52	0.9

Figure 4: Application speedups. Measurements were taken on a DECstation 5000/200, paging to a local RZ57 disk with a page size of 4 Kbytes. All benchmarks were run with approximately 14 Mbytes available for user processes. Compression was performed using Williams's LZRW1 algorithm. Some applications had a high fraction of pages compress less than the threshold (4:3), though the applications with the greatest speedup compressed well across all pages. Times are in minutes:seconds.

read from backing store. Ideally, one would use the compression cache in a system that permitted less than a 4-Kbyte read to satisfy a page fault, in which case Gold (and other applications) should benefit more generally from compression. Table "Application Speedup" lists three runs of *gold*:

- **gold_create**: This benchmark creates a new index from scratch. It has a high degree of write accesses, so the degradation it suffers by reading 4-Kbyte blocks is partly offset by writing compressed pages together to backing store. However, 42% of pages compress less than 4:3, and the average of the rest is only 59%. The program runs 11% more slowly with the compression cache than without.
- **gold_cold**: This benchmark performs a sequence of queries against an existing *gold* index engine, with the index engine having just started. Thus the index engine writes many pages as well as reading them. It runs 25% more slowly.
- **gold_warm**: Lastly, this benchmark performs the same set of queries once *gold_cold* has executed. The index data are already established in the address space of the index engine, and are faulted upon in a read-only fashion. A small number of pages are modified as the program operates, however. This benchmark runs 36% more slowly.

Obviously, for those applications that run 20-40% more slowly with the compression cache, varying the amount of memory is insufficient to prevent degradation. It should be possible to disable compression completely when poor compression is obtained.

6 Conclusions and Future Work

In conclusion, compression can reduce the amount of I/O to and from a backing store, possibly eliminating it completely. Even when I/O operations are still necessary, compressed pages require less bandwidth. Depending on the cost of compression, the cost of I/O, and the compressibility of memory pages, this technique can improve performance by factors of 3-4 or more in the best case.

However, "real" applications generally do not obtain this degree of improvement for a number of reasons:

- *locality*, which causes an application to take faults on compressed pages that would have been accessible in an unmodified system;
- *poor compressibility*, which results in less of a reduction in I/O for the same amount of effort; and
- *restricted I/O*, which causes larger transfers to be performed than are necessary.

One example of an application that *does* obtain significant speedup is a file comparison program that compresses well and whose sequential passes through a large two-dimensional array make it less susceptible to an increase in the fault rate. Other applications vary from moderate improvements in performance to slight (or even substantial) degradations. As compression gets faster relative to I/O, the range of applications that can benefit from compressed paging should improve. This can happen in any of several ways: hardware compression, which would improve the disparity between compression speeds and I/O rates; faster processors, which would do the same thing for software compression; and slower backing stores, such as wireless networks. A better interface to the backing store would help as well.

Note that the same techniques presented in this paper for virtual memory can potentially be applied to other areas as well. For instance, on systems with enough physical memory to make Sprite LFS practical, one might consider combining compressed Sprite LFS [4] with the compression cache techniques presented here: the system could keep part or all of the file buffer cache in compressed format in order to improve the cache hit rate. One might also redesign specific applications, such as databases, to keep some of their data structures in compressed format, using application-specific techniques for compressing data and managing the choice of data to compress. Experiences with the compression cache make it clear that the success of any scheme that uses compression to improve performance will depend a great deal on the relative speeds of compression and I/O, the compressibility of data, and data access patterns.

7 Acknowledgements

Brian Marsh and Rafael Alonso contributed to the design and initial implementation of the compression cache, and helped to formalize its expected performance. Krish Ponamgi helped with the implementation of the backing store. Cezary Dubnicki, Steven Johnson, Dan Lopresti, and Karin Petersen provided programs to help evaluate the compression cache.

Rafael Alonso, Lisa Bahler, Daniel Barbará, Brian Bershad, Jeff Esakov, Eben Haber, Hank Korth, Kai Li, Dick Lipton, Dan Lopresti, Brian Marsh, Krish Ponamgi, Sreedhar Sivarkumaran, Marvin Theimer, and Rosemary Walsh provided helpful feedback on earlier drafts of this paper, which helped improve its content and presentation. Lastly, I would like to thank Andrew Appel, Mike Burrows, Ramón Cáceres, Mike Jones, Karin Petersen, and Jonathan Sandberg for their comments and suggestions.

Bibliography

[1] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *Proceedings of the Fourth International Conference on*

Architectural Support for Programming Languages and Operating Systems, pages 96-107, Santa Clara, CA, April 1991.

[2] Russ Atkinson, Dan Greene, Bryan Lyles, and Marvin Theimer. Applying compression techniques to virtual memory paging. Xerox PARC Internal Memorandum, 1990.

[3] Daniel Barbará, Chris Clifton, Fred Douglis, Hector Garcia-Molina, Stephen Johnson, Ben Kao, Sharad Mehrotra, Jens Tellefsen, and Rosemary Walsh. The Gold mailer. In *9th International Conference on Data Engineering*, Vienna, April 1993. To appear.

[4] M. Burrows, C. Jerian, B. Lampson, and T. Mann. On-line data compression in a log-structured file system. In *The Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2-9. ACM, October 1992.

[5] Vincent Cate and Thomas Gross. Combining the concepts of compression and caching for a two-level filesystem. In *The Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 200-211. ACM, April 1991.

[6] C. Dubnicki and T. LeBlanc. Adjustable block size coherent caches. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 170-180, Gold Coast, Australia, May 1992. ACM.

[7] David B. Golub and Richard P. Draves. Moving the default memory manager out of the Mach kernel. In *Proceedings of the 2nd Mach Symposium*, pages 177-188, November 1991.

[8] R. J. Lipton and D. P. Lopresti. Comparing long strings on a short systolic array. In W. Moore, A. McCabe, and R. Urquhart, editors, *Systolic Arrays*, pages 181-190. Adam Hilger, Boston, 1987.

[9] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134-154, February 1988.

[10] M. N. Nelson. *Physical Memory Management in a Network Operating System*. PhD thesis, University of California, Berkeley, CA 94720, November 1988. Available as Technical Report UCB/CSD 88/471.

[11] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23-36, February 1988.

[12] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26-52, February 1992. Also appears in Proceedings of the 13th Symposium on Operating Systems Principles, October 1991.

[13] M. Schroeder, A. Birrell, M. Burrows, H. Murray, R. Needham, T. Rodeheffer, E. Satterthwaite, and C. Thacker. Autonet: A high-speed self-configuring local area network using point-to-point links. *IEEE Journal on Selected Areas in Communications*, 9(8):1318-1335, October 1991.

[14] Mark Taunton. Compressed executables: an exercise in thinking small. In *Proceedings of the USENIX 1991 Summer Conference*, 1991.

[15] Mark Weiser. The computer for the 21st century. *Scientific American*, pages 94-104, September 1991.

[16] Ross N. Williams. An extremely fast ZIV-Lempel data compression algorithm. In *Data Compression Conference*, pages 362-371, April 1991.

Author Information

Fred Douglis is a scientist at the Matsushita Information Technology Laboratory. His interests include mobile and distributed computing, file systems, and user interfaces. Prior to joining MITL in October 1991, he was a postdoctoral fellow at the Vrije Universiteit in Amsterdam, working on Amoeba and teaching a course in distributed systems. He worked on the Sprite network operating system from its inception in 1984 until the fall of 1990, and built its process migration facility as part of his doctoral research. He received a B.S. in Computer Science from Yale University in 1984, an M.S. in Computer Science from the University of California, Berkeley, in 1987, and a Ph.D. in Computer Science from U.C. Berkeley in 1990. Reach him electronically at douglis@mitl.com. Reach him via USMail at Matsushita Information Technology Laboratory; 182 Nassau Street; Princeton, NJ 08542 USA.

The USENIX Association

The UNIX and Advanced Computing Systems Professional and Technical Association

The USENIX Association is a not-for-profit membership organization of those individuals and institutions with an interest in UNIX and UNIX-like systems and, by extension, C++, X windows, and other programming tools. It is dedicated to:

- sharing ideas and experience relevant to UNIX or UNIX inspired and advanced computing systems,
- fostering innovation and communicating both research and technological developments,
- providing a neutral forum for the exercise of critical thought and airing of technical issues.

Founded in 1975, USENIX sponsors twice yearly general conferences accompanied by vendor displays and frequent single-topic conferences and symposia. USENIX publishes proceedings of its meetings, the bi-monthly newsletter ;login:, the refereed technical quarterly Computing Systems. (published with the University of California Press), and is expanding its publishing role in cooperation with MIT Press with a book series on advanced computing systems. The Association actively participates in various ANSI, IEEE and ISO standards efforts with a paid representative attending selected meetings News of standards efforts and reports of many meetings are reported in ;login:.

SAGE, the Systems Administrators' Guild

USENIX has recently launched its first Special Technical Groups (STGs), the Systems Administrators' Guild (SAGE). SAGE is devoted to the advancement of systems administration as a profession. It will recruit talented individuals to the profession, develop guidelines for the education of members of the profession, establish standards of professional excellence and provide recognition for those who attain them, and promote work that advances the state of the art and propagates knowledge of good practice in the profession.

USENIX and SAGE will work together to publish technical information and sponsor conferences, symposia , tutorials and local groups in the field of systems administration. Currently USENIX and SAGE jointly sponsor the annual Systems Administration Conference and they, together with FedUNIX, are sponsoring the 1993 World Conference on Tools and Techniques for Systems Administration, Networking and Security (SANS-II). SAGE News and other items of interest to systems administrators are found in each issue of the USENIX newsletter ;login:.

There are four classes of membership in the Association, differentiated primarily by the fees paid and services provided.

USENIX Association services include:

- Subscription to login:, a bi-monthly newsletter;
- Computing Systems, a refereed technical quarterly;
- Discounts on various UNIX and technical publications for purchase;
- Technical conference and tutorial program twice a year and single-topic symposia periodically;
- A discount on technical conference and workshop registration fees;
- The right to vote on matters affecting the Association, its bylaws, election of its directors and officers.
- Right to join Special Technical Groups such as SAGE

The supporting members of the USENIX Association are:

Open Software Foundation
UUNET Technologies, Inc.
UNIX System Laboratories, Inc.
Quality Micro Systems
Sun Micro Systems, Inc.
Sybase Inc.
mt Xinu
Frame Technology Corporation
Network Computing Devices, Inc.

For further information about membership, conferences or publications, contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710
Telephone: 510/528-8649
Email: office@usenix.org
FAX: 510/548-5738